**Patent Office**
**Canberra**

**CERTIFIED COPY OF PRIORITY DOCUMENT**

I, JONNE YABSLEY, TEAM LEADER EXAMINATION SUPPORT AND SALES hereby certify that annexed is a true copy of the Provisional specification in connection with Application No. PO 7991 for a patent by SILVERBROOK RESEARCH PTY LTD as filed on 15 July 1997.

WITNESS my hand this
Nineteenth day of December 2002

JONNE YABSLEY
TEAM LEADER EXAMINATION
SUPPORT AND SALES

THIS PAGE BLANK (USPTO)

P/00/009
Regulation 3.2

AUSTRALIA

Patents Act 1990

# PROVISIONAL SPECIFICATION

AUSTRALIAN
PROVISIONAL No. DATE OF FILING
P07991 15 JUL 97
PATENT OFFICE

Application Title: Image Processing Method and Apparatus (ART01)

The invention is described in the following statement:

GH REF: 23975X

# IMAGE PROCESSING METHOD AND APPARATUS (ART01)

## Field of the Invention

The present invention relates to an image processing method and apparatus and, in particular, discloses a Digital Instant Camera with Image Processing Capability.

The present invention further relates to the field of digital camera technology and, particularly, discloses a digital camera having an integral colour printer.

## Background of the Invention

Traditional camera technology has for many years relied upon the provision of an optical processing system which images a negative of an image onto a photosensitive film which is subsequently chemically processed so as to "fix" the film and to subsequently allow for positive prints to be produced which reproduce the original image. Such an image processing technology, although it has become a standard, can be unduly complexed, is expensive and difficult technologies are involved in full colour processing of images. Recently, digital cameras have become available. These cameras normally rely upon the utilisation of a charged coupled device (CCD) to image the particular image. The camera normally includes the storage media for the storage of the imaged devices in addition to a connector for the transfer of images to a subsequent computer device for manipulation and printing out.

Such devices are generally inconvenient in that all images must be stored by the camera and printed out at some later stage. Hence, the camera must have sufficient storage capabilities for the storing of multiple images and, additionally, the user of the camera must have access to a subsequent computer system for the downloading of the images and there printing out by a computer device or the like.

## Summary of the Invention

The present invention relates to providing an alternative form of camera system which includes a digital camera with an integral colour printer. Additionally, the

camera provides hardware and software for the increasing of the apparent resolution of the image sensing system and the conversion of the image to a wide range of "artistic styles" and a graphic enhancement.

In accordance with a first aspect of the present invention, there is provided a camera system comprising at least one area image sensor for imaging a scene, a camera processor means for processing said image scene in accordance with a predetermined scene transformation requirement, a printer for printing out said processed image scene on print media said printer, print media and printing ink stored in a single detachable module inside said camera system, said camera system comprising a portable hand held unit for the imaging of scenes by said area image sensor and printing said scenes directly out of said camera system via said printer.

Preferably the camera system includes a print roll for the storage of print media and printing ink for utilisation by the printer, s the aid print roll being detachable from the camera system. Further, the print roll can include an authentication chip containing authentication information and the camera processing means is adapted to interrogate the authentication chip so as to determine the authenticity of said print roll when inserted within said camera system.

Further, the printer can include a drop on demand ink printer and guillotine means for the separation of printed photographs.

Brief Description of the Drawings

Notwithstanding any other forms which may fall within the scope of the present invention, preferred forms of the invention will now be described, by way of example only, with reference to the accompanying drawings in which:

Fig. 1 illustrates and artcam device constructed in accordance with the preferred embodiment;

Fig. 2 is a schematic block diagram of the main Artcam electronic components;

Fig. 3 is a schematic block diagram of the Artcam Central Processor in more detail;

Fig. 4 illustrates the CCD image organisation;

Fig. 5 illustrates the storage format for a logical image;

Fig. 6 illustrates the internal image memory storage format;

Fig. 7 illustrates the image pyramid storage format;

Fig. 8 illustrates the process steps in creating an output image;

Fig. 9 illustrates the operation of an image iterator;

Fig. 10 illustrates an example read iterator;

Fig. 11 illustrates a standard process;

Fig. 12 illustrates an Iterator workload;

Fig. 13 illustrates a first example box read iterator output;

Fig. 14 illustrates a second example box read iterator output;

Fig. 15 illustrates a Box Read Iterator Process;

Fig. 16 illustrates the storage format utilised by a vertical strip Iterator;

Fig. 17 illustrates a process that requires only a vertical strip Write Iterator;

Fig. 18 illustrates the VLIW processor architecture;

Fig. 19 illustrates the I/O units block in more detail;

Fig. 20 illustrates the process of generating a sequential read;

Fig. 21 illustrates the internal portion of the sequential coordinate generator;

Fig. 22 illustrates the vertical strip generation process;

Fig. 23 illustrates an implementation of the vertical strip generation process;

Fig. 24 illustrates the form of a single CCD pixel;

Fig. 25 illustrates the CCD reading process;

Fig. 26 illustrates the process of sampling an artcard;

Fig. 27 illustrates the process of reading a rotated Artcard;

Fig. 28 illustrates a flow chart of the steps necessary to

decode an Artcard;

Fig. 29 illustrates a timeline of pixel reading of an Artcard;

Fig. 30 illustrates an enlargement of the left hand corner of a single Artcard;

Fig. 31 illustrates a single target for detection;

Fig. 32 illustrates the method utilised to detect targets;

Fig. 33 illustrates the method of calculating the distance between two targets;

Fig. 34 illustrates the process of centroid drift;

Fig. 35 shows one form of centroid lookup table;

Fig. 36 illustrates the centroid updating process;

Fig. 37 illustrates a delta processing lookup table utilised in the preferred embodiment;

Fig. 38 illustrates the process of unscrambling Artcard data;

Fig. 39 illustrates the convolution process;

Fig. 40 illustrates one form of implementation of the convolver;

Fig. 41 illustrates the compositing process;

Fig. 42 illustrates the regular compositing process in more detail;

Fig. 43 illustrates the process of warping using a warp map;

Fig. 44 illustrates the warping bi-linear interpolation process;

Fig. 45 illustrates the process of span calculation;

Fig. 46 illustrates the basic span calculation process;

Fig. 47 illustrates one form of detail implementation of the span calculation process;

Fig. 48 illustrates the process of reading image pyramid levels;

Fig. 49 illustrates using the pyramid table for bilinear interpolation;

Fig. 50 illustrates the histogram collection process;

Fig. 51 illustrates the color transform process;

Fig. 52 illustrates the color conversion process;

Fig. 53 illustrates the color space conversion process in more detail;

Fig. 54 illustrates the process of calculating an input coordinate;

Fig. 55 illustrates the basic process for calculating a pixel;

Fig. 56 illustrates the generalized scaling process;

Fig. 57 illustrates the scale in X scaling process;

Fig. 58 illustrates the scale in Y scaling process;

Fig. 59 illustrates the tessellation process;

Fig. 60 illustrates the sub-pixel translation process;

Fig. 61 illustrates the compositing process;

Fig. 62 illustrates the process of compositing with feedback;

Fig. 63 illustrates the process of tiling with color from the input image;

Fig. 64 illustrates the process of tiling with feedback;

Fig. 65 illustrates the process of tiling with texture replacement;

Fig. 66 illustrates the process of tiling with background and tile texture;

Fig. 67 illustrates the process of applying a texture without feedback;

Fig. 68 illustrates the process of applying a texture with feedback;

Fig. 69 illustrates the process of rotation of CCD pixels;

Fig. 70 illustrates the process of interpolation of Green subpixels;

Fig. 71 illustrates the process of interpolation of Blue subpixels;

Fig. 72 illustrates the process of interpolation of Red subpixels;

Fig. 73 illustrates the process of CCD pixel interpolation with O degree rotation for odd pixel lines;

Fig. 74 illustrates the process of CCD pixel interpolation with 0 degree rotation for even pixel lines;

Fig. 75 illustrates the process of color conversion to Lab color space;

Fig. 76 illustrates the process of calculation of $1/\sqrt{X}$;

Fig. 77 illustrates the implementation of the calculation of $1/\sqrt{X}$ in more detail;

Fig. 78 illustrates the process of Normal calculation with a bump map;

Fig. 79 illustrates the process of illumination calculation with a bump map;

Fig. 80 illustrates the process of illumination calculation with a bump map in more detail;

Fig. 81 illustrates the process of calculation of L using a directional light;

Fig. 82 illustrates the process of calculation of L using a Omni lights and spotlights;

Fig. 83 illustrates one form of implementation of calculation of L using a Omni lights and spotlights;

Fig. 84 illustrates the process of calculating the N.L dot product;

Fig. 85 illustrates the process of calculating the N.L dot product in more detail;

Fig. 86 illustrates the process of calculating the R.V dot product;

Fig. 87 illustrates the process of calculating the R.V dot product in more detail;

Fig. 88 illustrates the attenuation inputs and outputs;

Fig. 89 illustrates an actual implementation of attenuation calculation;

Fig. 90 illustrates a graph of the cone factor;

Fig. 91 illustrates the process of penumbra calculation;

Fig. 92 illustrates the angles utilised in penumbra calculation;

Fig. 93 illustrates the inputs and outputs to penumbra calculation;

Fig. 94 illustrates an actual implementation of penumbra calculation;

Fig. 95 illustrates the inputs and outputs to ambient calculation;

Fig. 96 illustrates an actual implementation of ambient calculation;

Fig. 97 illustrates an actual implementation of diffuse calculation;

Fig. 98 illustrates the inputs and outputs to a diffuse calculation;

Fig. 99 illustrates an actual implementation of a diffuse calculation;

Fig. 100 illustrates the inputs and outputs to a specular calculation;

Fig. 101 illustrates an actual implementation of a specular calculation;

Fig. 102 illustrates the inputs and outputs to a specular calculation;

Fig. 103 illustrates an actual implementation of a specular calculation;

Fig. 104 illustrates an actual implementation of a ambient only calculation;

Fig. 105 illustrates the process overview of light calculation;

Fig. 106 illustrates an example illumination calculation for a single infinite light source;

Fig. 107 illustrates an example illumination calculation for a Omni light source without a bump map;

Fig. 108 illustrates an example illumination calculation for a Omni light source with a bump map;

Fig. 109 illustrates an example illumination calculation for a Spotlight light source without a bump map;

Fig. 110 illustrates the process of applying a single Spotlight onto an image with an associated bump-map;

Fig. 111 illustrates the logical layout of a single printhead;

Fig. 112 illustrates the structure of the printhead interface;

Fig. 113 illustrates the process of rotation of an Lab image;

Fig. 114 illustrates the format of a printed image;

Fig. 115 illustrates the dithering process;

Fig. 116 illustrates the process of generating an 8 bit dot output;

Fig. 117 illustrates a card reader;

Fig. 118 illustrates an exploded perspective of a card reader;

Fig. 119 illustrates a closeup view of the Artcard reader;

Fig. 120 illustrates a perspective view of the print roll and print head;

Fig. 121 illustrates a first exploded perspective view of the print roll;

Fig. 122 illustrates a second exploded perspective view of the print roll;

Fig. 123 illustrates the print roll authentication chip;

Fig. 124 illustrates an enlarged view of the print roll authentication chip;

Fig. 125 illustrates the architecture of the print roll authentication chip;

Fig. 126 sets out the information stored on the print roll authentication chip;

Fig. 127 illustrates the authentication process upon insertion of a print roll;

Fig. 128 illustrates a shielding metal layer placed on top of the authentication chip;

Fig. 129 illustrates the data stored within the Artcam authorisation chip;

Fig. 130 illustrates the process of print head pulse characterisation;

Fig. 131 is an exploded perspective, in section, of the print head ink supply mechanism;

Fig. 132 is a bottom perspective of the ink head supply unit;

Fig. 133 is a bottom side sectional view of the ink head

supply unit;

Fig. 134 is a top perspective of the ink head supply unit;

Fig. 135 is a top side sectional view of the ink head supply unit;

Fig. 136 illustrates a wire frame view of a small portion of the print head;

Fig. 137 illustrates is an exploded perspective of the print head unit;

Fig. 138 illustrates a top side perspective view of the internal portions of an Artcam camera, showing the parts flattened out;

Fig. 139 illustrates a bottom side perspective view of the internal portions of an Artcam camera, showing the parts flattened out;

Fig. 140 illustrates a first top side perspective view of the internal portions of an Artcam camera, showing the parts as encased in an Artcam;

Fig. 141 illustrates a second top side perspective view of the internal portions of an Artcam camera, showing the parts as encased in an Artcam;

Fig. 142 illustrates a second top side perspective view of the internal portions of an Artcam camera, showing the parts as encased in an Artcam;

Fig. 143 illustrates the structure of the ALUs block;

Fig. 144 illustrates the structure of the read unit;

Fig. 145 illustrates the structure of the write unit;

Fig. 146 illustrates the structure of the ReadWrite unit;

Fig. 147 illustrates the structure of the Adder ALU;

Fig. 148 illustrates the structure of the Multiply ALU;

Fig. 149 illustrates the structure of the Logical ALU;

Fig. 150 illustrates the structure of the Display Controller;

Fig. 151 illustrates the backing portion of a postcard print roll;

Fig. 152 illustrates the corresponding front image on the postcard print roll after printing out images;

Fig. 153 illustrates a form of print roll ready for purchase by a consumer.

Description of preferred and other Embodiments

The digital image processing camera system constructed in accordance with the preferred embodiment is as illustrated in Fig. 1. A digital image in camera 1 is provided and includes means for the insertion of an integral print roll (not shown). The camera unit 1 can include an area image sensor 2 which sensors an image 3 for captured by the camera. Optionally, the second area image sensor 4 can be provided to also image the scene 3 and to optionally provide for the production of stereographic output effects.

The camera 1 can include an optional colour display 5 for the display of the image being sensed by the sensor 2. When a simple image is being displayed on the display 5, the button 6 can be depressed resulting in the printed image 8 being output by the camera unit 1. A series of cards, herein after known as "artcards" 9 containing, on one surface encoded information and on the other surface, containing an image distorted by the particular effect produced by artcard 9. The artcard 9 is inserted in an artcard reader 10 in the back of camera 1 and, upon the insertion, results in output image 8 being distorted in the same manner as the distortion appearing on the surface of artcard 9. Hence, a user wishing to produce a particular effect can insert one of many artcards 9 into the artcard reader 10 and utilise button 6 to take a picture of the image 3 resulting in a corresponding distorted output image 8.

The camera unit 1 can also include a number of other control button 13, 14 in addition to a simple LCD output display 15 for the display of informative information including the number of printouts left on the internal print roll on the camera unit.

Turning now to Fig. 2, there is illustrated a schematic

view of the internal hardware of the camera unit 1. The internal hardware is based around an Artcam central processor unit (ACP) 31.

Artcam Central Processor 31

The Artcam central processor 31 provides many functions which form the 'heart' ofthe system. The ACP 31 is preferably implemented as a complex, high speed, CMOS system on-a-chip. Utilising standard cell design with some full custom regions is recommended. Fabrication on a 0.25µ CMOS process will provide the density and speed required, along with a reasonably small die area.

The functions provided by the ACP 31 include:

1. Control and digitisation of the area image sensor 2. A 3D stereoscopic version of the ACP requires two area image sensor interfaces with a second optional image sensor 4 being provided for stereoscopic effects.

2. Area image sensor compensation, reformatting, and image enhancement.

3. Memory interface and management to a memory store 33.

4. Interface, control, and analog to digital conversion of an Artcard reader linear image sensor 34 which is provided for the reading of data from the artcards 9.

5. Extraction of the raw Artcard data from the digitised and encoded Artcard image.

6. Reed-Solomon error detection and correction of the Artcard encoded data. The encoded surface of the artcard 9 includes information on how to process an image to produce the effects displayed on the image distorted surface of the artcard 9. This information is in the form of a script, hereinafter known as a "Vark script". The Vark script is utilised by an interpreter running within the ACP 31 to produce the desired effect.

7. Interpretation of the Vark script on the Artcard 9.

8. Performing image processing operations as specified by the Vark script.

9. Controlling various motors for the paper transport 36, zoom lens 38, autofocus 39 and Artcard driver 37.

10. Controlling a guillotine actuator 40 for the operation of a guillotine 41 for the cutting of photographs 8 from print roll 42.

11. Half-toning of the image data for printing.

12. Providing the print data to a printhead 44 at the appropriate times.

13. Controlling the print head 44.

14. Controlling the ink pressure feed to printhead 44.

15. Controlling optional flash unit 56.

16. Reading and acting on various sensors in the camera, including camera orientation sensor 46, autofocus 47 and Artcard insertion sensor 49.

17. Reading and acting on the user interface buttons 6, 13, 14.

18. Controlling the status display 15.

19. Providing viewfinder and preview images to the colour display 5.

20. Control of the system power consumption, including the ACP power consumption via power management circuit 51 .

21. Providing external communications 52 to general purpose computers (using USB).

22. Reading and storing information in a printing roll authentication chip 53.

23. Reading and storing information in a camera authentication chip 54.

24. Communicating with an optional mini-keyboard 57 for text modification.

Quartz crystal 58

A quartz crystal 58 is used as a frequency reference for the system clock. As the system clock is very high, the ACP 31 includes a phase locked loop clock circuit to increase the frequency derived from the crystal 58.

Image Sensing

Area image sensor 2

The area image sensor 2 converts an image through its lens into an electrical signal. It can either be a charge coupled device (CCD) or an active pixel sensor (APS)CMOS image sector. At present, available CCD's normally have a higher image quality, however, there is currently much development occurring in CMOS imagers. CMOS images are eventually expected to be substantially cheaper than CCD's have smaller pixel areas, and be able to incorporate drive circuitry and signal processing. They can also be made in CMOS fabs, which are transitioning to 12" wafers. CCD's are usually built in 6" wafer fabs, and economics may not allow a conversion to 12" fabs. Therefore, the difference in fabrication cost between CCD's and CMOS imagers is likely to increase, progressively favouring CMOS imagers. However, at present, a CCD is probably the best option.

The Artcam unit will produce suitable results with a 1,500 x 1,000 area image sensor. However, smaller sensors, such as 750 x 500, will be adequate for many markets. The Artcam is less sensitive to image sensor resolution than are conventional digital cameras. This is because many of the styles contained on Artcards 9 process the image in such a way as to obscure the lack of resolution. For example, if the image is distorted to simulate the effect of being converted to an impressionistic painting, low source image resolution can be used with minimal effect. Further examples for which low resolution input images will typically not be noticed include image warps which produce high distorted images, multiple miniature copies of the of the image (eg. passport photos), textural processing such as bump mapping for a base relief metal look, and photo-compositing into structured scenes.

This tolerance of low resolution image sensors may be a significant factor in reducing the manufacturing cost of an

Artcam unit 1 camera. An Artcam with a low cost 750 x 500 image sensor will often produce superior results to a conventional digital camera with a much more expensive 1,500 x 1,000 image sensor.

Optional stereoscopic 3D image sensor 4

The 3D versions of the Artcam unit 1 have an additional image sensor 4, for stereoscopic operation. This image sensor is identical to the main image sensor. The circuitry to drive the optional image sensor may be included as a standard part of the ACP chip 31 to reduce incremental design cost. Alternatively, a separate 3D Artcam ACP can be designed. This option will reduce the manufacturing cost of a mainstream single sensor Artcam.

Print roll authentication chip 53

A small chip 53 is included in each print roll 42. This chip replaced the functions of the bar code, optical sensor and wheel, and ISO/ASA sensor on other forms of camera film units such as Advanced Photo Systems file cartridges.

The authentication chip also provides other features:

1. The storage of data than is mechanically and optically sensed from APS rolls

2. A remaining media length indication, accurate to mm.

3. Authentication Information to prevent inferior copies.

The authentication chip 53 contains 1024 bits of Flash memory, of which 128 bits is an authentication key, and 512 bits is the authentication information. Also included is an encryption circuit to ensure that the authentication key cannot be accessed directly.

Printhead 44

The Artcam unit 1 can utilise any colour print technology which is small enough, low enough power, fast enough, high enough quality, and low enough cost, and is compatible with the print roll. Relevant printheads will be

specifically discussed hereinafter.

The specifications of the ink jet head are:

| | |
|---|---|
| Image type | Bi-level, dithered |
| Colour | CMY Process Colour |
| Resolution | 1600 dpi |
| Print head length | 'Page-width' (100mm) |
| Print speed | 2 seconds per photo |

Optional ink pressure Controller (not shown)

The function of the ink pressure controller depends upon the type of ink jet print head 44 incorporated in the Artcam. For some types of ink jet, the use of ink pressure controller can be eliminated, as the ink pressure is simply atmospheric pressure. Other types of print head require a regulated positive ink pressure. In this case, the in pressure controller consists of a pump and pressure transducer.

Other print heads may require an ultrasonic transducer to cause regular oscillations in the ink pressure, typically at frequencies around 100KHz. In the case, the ACP 31 controls the frequency phase and amplitude of these oscillations.

Paper transport motor 36

The paper transport motor 36 moves the paper from within the print roll 42 past the print head as a relatively constant rate. The motor 36 is a miniature motor geared down to an appropriate speed to drive rollers which move the paper. A high quality motor and mechanical gears are required to achieve high image quality, as mechanical rumble or other vibrations will affect printed dot row spacing.

Paper transport motor driver 60

The motor driver 60 is a small circuit which amplified the digital motor control signals from the APC 31 to levels suitable for driving the motor 36.

Paper pull sensor

A paper pull sensor 50 detects a user's attempt to pull a photo from the camera unit during the printing process. The APC 31 reads this sensor 50, and activates the guillotine 41 if the condition occurs. The paper pull sensor 50 is incorporated to make the camera more 'foolproof' in operation. Were the user to pull the paper out forcefully during printing, the print mechanism 44 or print roll 42 may (in extreme cases) be damaged. Since it is acceptable to pull out the 'pod' from a Polaroid type camera before it is fully ejected, the public has been 'trained' to do this. Therefore, they are unlikely to heed printed instructions not to pull the paper.

The Artcam preferably restarts the photo print process after the guillotine 41 has cut the paper after pull sensing.

The pull sensor can be implemented as a strain gauge sensor, or as an optical sensor detecting a small plastic flag which is deflected by the torque that occurs on the paper drive rollers when the paper is pulled. The latter implementation is recommendation for low cost.

Paper guillotine actuator

The paper guillotine actuator 40 is a small actuator which causes the guillotine 41 to cut the paper either at the end of a photograph, or when the paper pull sensor 50 is activated.

Paper guillotine actuator driver 40

The guillotine actuator drive 40 is a small circuit which amplifies a guillotine control signal from the APC tot the level required by the actuator 41.

Artcard 9

The Artcard 9 is a program storage medium for the Artcam unit. As noted previously, the programs are in the form of Vark scripts. Vark is a powerful image processing language especially developed for the Artcam unit. Each Artcard 9 contains one Vark script, and thereby defines one image processing style.

Preferably, the VARK language is highly image processing specific. By being highly image processing specific, the amount of storage required to store the details of the card are substantially reduced. Further, the ease with which new programs can be created, including enhanced effects, is also substantially increased. Preferably, the language includes facilities for handling many image processing functions including image warping via a warp map, convolution, color lookup tables, posterizing an image, adding noise to an image, image enhancement filters, painting algorithms, brush jittering and manipulation edge detection filters, tiling, illumination via light sources, bumpmaps, text, face detection and object detection attributes, fonts, including three dimensional fonts, and arbitrary complexity pre-rendered icons.

Attached in Appendix D is an example of the VARK language which includes all of these facilities and has been defined by the present applicant with image processing functionality in mind.

Hence, by utilizing the language constructs as defined by the created language, new affects on arbitrary images can be created and constructed for inexpensive storage on Artcard and subsequent distribution to camera owners. Further, on one surface of the card can be provided an example illustrating the effect that a particular VARK script, stored on the other surface of the card, will have on an arbitrary captured image.

By utilizing such a system, camera technology can be distributed without a great fear of obsolescence in that, provided a VARK interpreter is incorporated in the camera device, a device independent scenario is provided whereby the underlying technology can be completely varied over time. Further, the VARK scripts can be updated as new filters are created and distributed in an inexpensive manner, such as via simple cards for card reading.

The Artcard 9 is a piece of thin white plastic with the

same format as a credit card (86mm long by 54mm wide). The Artcard is printed on both sides using a high resolution ink jet printer. The inkject printer technology is assumed to be the same as that used in the Artcam, with 1600 dpi (63dpmm) resolution. A major feature of the Artcard 9 is low manufacturing cost. Artcards can be manufactured at high speeds as a wide web of plastic film. The plastic web is coated on both sides with a hydrophilic dye fixing layer. The web is printed simultaneously on both sides using a 'pagewidth' colour ink jet printer. The web is then slit and punched into individual cards. On one face of the card is printed a human readable representation of the effect the Artcard 9 will have on the sensed image. This can be simply a standard image which has been processed using the Vark script stored on the back face of the card.

On the back face of the card is printed an array of dots which can be decoded into the Vark script that defines the image processing sequence. The print area is 80mm x 50mm, giving a total of 15,876,000 dots. This array of dots could represent at least 1.89 Mbytes of data. To achieve high reliability, extensive error detection and correction is incorporated in the array of dots. This allows a substantial portion of the card to be defaced, worn, creased, or dirty with no effect on data integrity. The data coding used is Reed-Solomon coding, with half of the data devoted to error correction. This allows the storage of 967 Kbytes of error corrected data on each Artcard 9.

Linear image sensor 34

The Artcard linear sensor 34 converts the aforementioned Artcard data image to electrical signals. As with the area image sensor 2, 4, the linear image sensor can be fabricated using either CCD or APS CMOS technology. The active length of the image sensor 34 is 50mm, equal to the width of the data array on the Artcard 9. To satisfy Nyquist's sampling theorem, the resolution of the linear image sensor 34 must be at least twice the highest spatial

frequency of the Artcard optical image reaching the image sensor. In practice, data detection is easier if the image sensor resolution is substantially above this. A resolution of 4800 dpi (189 dpmm) is chosen, giving a total of 9,450 pixels. This resolution requires a pixel sensor pitch of 5.3µm. This can readily be achieved by using four staggered rows of 20µm pixel sensors.

The linear image sensor is mounted in a special package which includes a LED 65 to illuminate the Artcard 9 via a light-pipe (not shown).

The Artcard reader light-pipe can be a moulded light-pipe which has several function:

1. It diffuses the light from the LED over the width of the card using total internal reflection facets.

2. It focuses the light onto a 16µm wide strip of the Artcard 9 using an integrated cylindrical lens.

3. It focuses light reflected from the Artcard onto the linear image sensor pixels using a moulded array of microlenses.

Artcard reader motor 37

The Artcard reader motor propels the Artcard past the linear image sensor 34 at a relatively constant rate. As it may not be cost effective to include extreme precision mechanical components in the Artcard reader, the motor 37 is a standard miniature motor geared down to an appropriate speed to drive a pair of rollers which move the Artcard 9. The speed variations, rumble, and other vibrations will affect the raw image data as circuitry within the APC 31 includes extensive compensation for these effects to reliably read the Artcard data.

The motor 37 is driven in reverse when the Artcard is to be ejected.

Artcard motor driver 61

The Artcard motor driver 61 is a small circuit which amplifies the digital motor control signals from the APC 31

to levels suitable for driving the motor 37.

Card Insertion sensor 49

The card insertion sensor 49 is an optical sensor which detects the presence of a card as it is being inserted in the card reader 34. Upon a signal from this sensor 49, the APC 31 initiates the card reading process, including the activation of the Artcard reader motor 37.

Card eject button 13

A card eject button 13 (Fig. 1) is used by the user to eject the current Artcard, so that another Artcard can be inserted. The APC 31 detects the pressing of the button, and reverses the Artcard reader motor 37 to eject the card.

Card status indicator 66

A card status indicator 66 is provided to signal the user as to the status of the Artcard reading process. This can be a standard bi-colour (red/green) LED. When the card is successfully read, and data integrity has been verified, the LED lights up green continually. If the card is faulty, then the LED lights up red.

If the camera is powered from a 1.5 V instead of 3V battery, then the power supply voltage is less than the forward voltage drop of the greed LED, and the LED will not light. In this case, red LEDs can be used, or the LED can be powered from a voltage pump which also powers other circuits in the Artcam which require higher voltage.

64 Mbit DRAM 33

To perform the wide variety of image processing effects, the camera utilises 8 Mbytes of memory 33. This can be provided by a single 64 Mbit memory chip. Of course, with changing memory technology increased Dram storage sizes may be substituted.

High speed access to the memory chip is required. This can be achieved by using a Rambus DRAM (burst access rate of 500 Mbytes per second) or chips using the new open standards such as double data rate (DDR) SDRAM or Synclink DRAM.

Camera authentication chip

The camera authentication chip 54 is identical to the print roll authentication chip 53, except that it has different information stored in it. The camera authentication chip 54 has three main purposes:

1. To provide a secure means of comparing authentication codes with the print roll authentication chip;

2. To provide storage for manufacturing information, such as the serial number of the camera;

3. To provide a small amount of non-volatile memory for storage of user information.

Displays

The Artcam includes an optional colour display 5 and small status display 15. Lowest cost consumer cameras may include a colour image display, such as a small TFT LCD 5 similar to those found on some digital cameras and camcorders. The colour display 5 is a major cost element of these versions of Artcam, and the display 5 plus back light are a major power consumption drain.

Status display 15

The status display 15 is a small passive segment based LCD, similar to those currently provided on silver halide and digital cameras. Its main function is to show the number of prints remaining in the print roll 42 and icons for various standard camera features, such as flash and battery status.

Colour display 5

The colour display 5 is a full motion image display which operates as a viewfinder, as a verification of the image to be printed, and as a user interface display. The cost of the display 5 is approximately proportional to its area, so large displays (say 4" diagonal) unit will be restricted to expensive versions of the Artcam unit. Smaller displays, such as colour camcorder viewfinder TFT's at around 1", may be effective for mid-range Artcams.

Zoom lens (not shown)

The Artcam can include a zoom lens. This can be a standard electronically controlled zoom lens, identical to one which would be used on a standard electronic camera, and similar to pocket camera zoom lenses. A referred version of the Artcam unit may include standard interchangeable 35mm SLR lenses.

Autofocus motor 39

The autofocus motor 39 changes the focus of the zoom lens. The motor is a miniature motor geared down to an appropriate speed to drive the autofocus mechanism.

Autofocus motor driver 63

The autofocus motor driver 63 is a small circuit which amplifies the digital motor control signals from the APC 31 to levels suitable for driving the motor 39.

Zoom motor 38

The zoom motor 38 moves the zoom front lenses in and out. The motor is a miniature motor geared down to an appropriate speed to drive the zoom mechanism.

Zoom motor driver 62

The zoom motor driver 62 is a small circuit which amplifies the digital motor control signals from the APC 31 to levels suitable for driving the motor.

Communications

The ACP 31 contains a universal serial bus (USB) interface 52 for communication with personal computers. Not all Artcam models are intended to include the USB connector, as an added means of differentiating low end Artcams from up-market models. However, the silicon area required for a USB circuit 52 is small, so the interface can be included in the standard ACP.

Optional Keyboard 57

The Artcam unit may include an optional miniature keyboard 57 for customising text specified by the Artcard. Any text appearing in an Artcard image may be editable, even if it is in a fancy metallic 3D font. The miniature keyboard includes a single line alphanumeric LCD to display

the original text and edited text. The keyboard may be a standard accessory.

The ACP 31 contains a serial communications circuit for transferring data to and from the miniature keyboard.

Power Supply

The Artcam unit uses a single battery 48. Depending upon the Artcam options, this is either a 3V Lithium cell, or a 1.5 VAA or AAA alkaline cell.

Power Management Unit 51

Power consumption is an important design constraint in the Artcam. It is desirable that either standard camera batteries (such as 3V lithium batters) or standard AA or AAA alkaline cells can be used. While the electronic complexity of the Artcam unit is dramatically higher than 35mm photographic cameras, the power consumption need not be commensurately higher. Power in the Artcam can be carefully managed with all unit being turned off when not in use.

The most significant current drains are the ACP 31, the area image sensors 2,4, the printer 44 various motors, the flash unit, 45 and the optional colour display 5 (if included) dealing with each part separately:

1. ACP: If fabricated using 0.25µm CMOS, and running on 1.5V, the ACP power consumption can be quite low. Clocks to various parts of the ACP chip can be quite low. Clocks to various parts of the ACP chip can be turned off when not in use, virtually eliminating standby current consumption. The ACP will only fully used for approximately 4 seconds for each photograph printed.

2. Area image sensor: power is only supplied to the area image sensor when the user has their finger on the button.

3. The printer power is only supplied to the printer when actually printing. This is for around 2 seconds for each photograph. Even so, suitably lower power consumption printing should be used.

4. The motors required in the Artcam are all low power miniature motors, and are typically only activated for a few seconds per photo.

5. The flash unit 45 is only used for some photographs. Its power consumption can readily be provided by a 3V lithium battery for a reasonably battery life.

6. The optional colour display 5 is a major current drain for two reasons: it must be on for the whole time that the camera is in use, and a backlight will be required if a liquid crystal display is used. Cameras which incorporate a colour display will require a larger battery to achieve acceptable batter life.

Flash unit 45

The flash unit 45 can be a standard miniature electronic flash for consumer cameras.

Artcam Central Processor

Turning now to Fig. 3, there is illustrated the Artcam central processor 31 in more detail. The ACP 31 can take many different forms depending on the technologies utilised. One for of ACP 31 is will now be described and includes the following components:

Image Address Interface 93

Images are manipulated within the Artcam in a variety of ways. Some methods of manipulation require random access to pixels within an image, while others require access to pixels in a specific logical order. The Image Address Interface provides an interface between a client and the cached DRAM, allowing specific known processing orders to be appropriately cached.

The DRAM interface 81 includes 128 cached lines, each 32 bytes wide (32 bytes being the standard Rambus data transfer unit).

The total memory on chip for caches is therefore 4096 bytes (128 x 32 bytes). The break up of cache assignment is:

-16 to cache the CPU's program (so programs can run at the same time as control ACP processes)

-16 to cache CPU program's data

-96 floating. These can be assigned to ALUs for particular functions, or assigned to CPU program or data as desired.

The 128 cache lines are divided into 8 groups of 16 for separate addressing in a given cycle, with appropriate multiplexing.

As stated previously, the image address interface is responsible for interfacing between other client portions of the ACP chip and the RAMBUS DRAM. In effect, each module within the image address interface (IAI) 93 is an address generator.

There are basically 3 logical types of images manipulated by the ACP. They are:

-CCD Image, which is the Input Image captured from the CCD.

-Internal Image format - the Image format utilsed interanly by the Artcam device.

Print Image - the Output Image format printed by the Artcam

These images are typically different in colour space, resolution, and the output & input colour spaces can vary from camera to camera. For example, a CCD image on a low-end camera may be a different resolution, or have different colour characteristics from that used in a high-end camera. However all internal image formats are the same format in terms of colour space across all cameras.

In addition, the 3 image types can vary with respect to which direction is 'up'. The physical orientation of the camera causes the notion of a portrait or landscape image, and this must be maintained throughout processing. For this reason, the internal image is always oriented correctly, and rotation is performed on images obtained from the CCD and during the print operation.

CCD Image Organisation

Although many different CCD image sensors could be

utilised, it will be assumed that the CCD itself is a 750 x 500 image sensor, yielding 375,000 bytes (8 bits per pixel). Each 2x2 pixel block having the following configuration as depicted in Fig.4.

A CCD Image as stored in DRAM has consecutive pixels from a given line contiguous in memory. Each line is stored one after the other. The image sensor interface (ISI) 83 is responsible for taking data from the CCD and storing it in the DRAM correctly oriented. Thus a CCD image with rotation 0 degrees has its first line G, R, G, R, G, R... and its second line as B, G, B, G, B, G... If the CCD image should be portrait, rotated 90 degrees, the first line will be R, G, R, G, R, G and the second line G, B, G, B, G, B...etc.

Pixels are stored in an interleaved fashion since all colour components are required in order to convert to the internal image format.

It should be noted that the ACP 31 makes no assumptions about the CCD pixel format, since actual CCDs for imaging may vary from Artcam to Artcam, and over time. All processing that takes place via the hardware is controlled by microcode in an attempt to extend the usefulness of the ACP 31.

Internal Image Organisation

Internal images typically consist of a number of channels. Vark images can include, but are not limited to:

Lab

Lab$\alpha$

Lab$\beta$

$\alpha\beta$

L

L, a and b correspond to components of the Lab colour space, $\alpha$ is a matte channel (used for composing), and $\beta$ is a bump-map channel (used during brushing & illuminating).

The Vark Accelerator 79 functions require images to be organised in a planar configuration. Thus a Lab image would

be stored as 3 separate (probably contiguous) blocks of memory:

one block for the L channel,

one block for the a channel, and

one block for the b channel

Within each channel block, pixels are stored contiguously for a given row (plus some optional padding bytes), and rows are stored one after the other.

Turning to Fig.5 there is illustrated an example form of storage of a logical image 100. The logical image 100 is stored in a planar fashion having L 101, a 102 and b 103 colour components stored one after another. Alternatively, the logical image 100 can be stored in a compressed format having an uncompressed L component 101 and compressed A and B components 105, 106.

Turning to Fig. 6, the pixels of for line n 110 are stored together before the pixels of for line and n + 1 (111). With the image being stored in contiguous memory within a single channel.

In the 8MB-memory model, the final Print Image after all processing is finished, needs to be compressed in the chrominance channels. Compression of chrominance channels is 4:1, causing an overall compression of 12:6, or 2:1.

Other than the final Print Image, images in the Artcam are typically not compressed. Because of memory constraints, software may choose to compress the final Print Image in the chrominance channels by scaling each of these channels by 2:1. If this has been done, the PRINT Vark function call utilised to print an image must be told to treat the specified chrominance channels as compressed. The PRINT function is the only function that knows how to deal with compressed chrominance, and even so, it only deals with a fixed 2:1 compression ratio.

Although it is possible to compress an image and then operate on the compressed image to create the final print image, it is not recommended due to a loss in resolution. In

addition, an image should only be compressed once - as the final stage before printout. While one compression is virtually undetectable, *multiple* compressions may cause substantial image degradation.

Clip image Organisation

Clip images stored on Artcards have no explicit support by the ACP 31. Software is responsible for taking any images from the current Artcard and organising the data into a form known by the ACP. If images are stored compressed on an Artcard, software is responsible for decompressing them, as there is no specific hardware support for decompression of Artcard images.

Image Pyramid Organisation

During brushing, tiling, and warping processes utilised to manipulate an image it is necessary to compute the average colour of a particular area in an image. Rather than calculate the value for each area given, these functions make use of an image pyramid. As illustrated in Fig.7, an image pyramid is effectively a multi-resolution pixel-map. The original image 115 is a 1:1 representation. Low-pass filtering and sub-sampling by 2:1 in each dimension produces an image ¼ the original size 116. This process continues until the entire image is represented by a single pixel. An image pyramid is constructed from an original internal format image, and consumes 1/3 of the size taken up by the original image (1/4 + 1/16 + 1/64 + ...). For an original image of 1500 x 1000 the corresponding image pyramid is approximately ½MB. An image pyramid is constructed by a specific Vark function, and is used as a parameter to other Vark functions.

Print Image Organisation

The entire processed image is required at the same time in order to print it. However the Print Image output can comprise a CMY dithered image is only a transient image format, used within the Print Image functionality. However, it should be noted that colour conversion will need to take

place from the internal colour space to the print colour space. In addition, this colour conversion can be tuned to be different for different print rolls in the camera with different ink characteristics e.g. Sepia output can be accomplished by using a specific sepia toning Artcard, or by using a sepia tone print-roll (so all Artcards will work in sepia tone).

Colour Spaces

There are 3 colour spaces used in the Artcam, corresponding to the different image types:

CCD Image has a unique CCD colour space

Internal Image has the internal colour space

Print Image has the printer colour space

The ACP has no direct knowledge of specific colour spaces. Instead, it relies on client colour space conversion tables to convert between CCD, internal, and printer colour spaces:

CCD RGB

Internal Lab

Printer CMY

Removing the colour space conversion from the ACP 31 allows:

-Different CCDs to be used in different cameras

-Different inks (in different print rolls over time) to be used in the same camera

-Separation of CCD selection from ACP design path

-A well defined internal colour space for accurate colour processing

The overall process for creating an output image is as illustrated in Fig.8. The process 120 includes rigging in a CCD image 121 in a CCD colour space, the conversion of the CCD image to an internal image 122 in an internal colour space, the continual processing 123 of the internal image to produce a final internal image, followed by the creation of a print image 124 for printing out in the printer's colour space. With each conversion 126, 127 colour tables are

required for colour mapping the images from one colour space to another. These colour tables can be provided in the Artcam ROM or in the particular print ROM.

Image access

Access to images is via special image address generators, defined logically below. The Image Address Interface 93 contains a number of these address generation state machines (AGSM).

Each AGSM has a set of registers for defining image characteristics:

| Register Name | # bits | Description |
|---|---|---|
| ImageStart | 32 | The address in memory where the image starts |
| ImageHeight | 12 | The number of lines in the image |
| ImageWidth | 12 | The number of pixels in a line |
| RowOffset | 12 | The number of bytes from one row to the next.<br>Equals ImageWidth + any padding |
| StartRow | 12 | Which row to start at in the image |
| EndRow | 12 | The last row+1 to be returned or written to within the image |
| StartPixel | 12 | Left border of the section of the image |
| EndPixel | 12 | The last pixel+1 to be returned or written to along a given row. |
| Loop | 1 | Keep looping the data. |

Random Access to pixels

Images are rarely required to be accessed in completely random (x, y) fashion, although it is straightforward enough to access a given pixel within a channel by the following addressing algorithm:

Address for pixel (X, Y) = ImageStart + (RowOffset * Y) + X.

This only gives the address of a single colour channel's component, and 3 such operations would be required to access all 3 colour components of a single pixel.

Image Iterators = Sequential Access to pixels

The primary image pixel access method for software and hardware algorithms is via Image Iterators located within the Image Address Interface 93. Image iterators perform all of the addressing and caching of the pixels within an image channel and either read or write pixels for their client. Read Iterators read pixels in a specific order for their clients, and Write Iterators write pixels in a specific order for their clients.

Turning to Fig.9, there is illustrated the operation of the Image Iterators of the embodiment. Each iterator, e.g 130, is interconnected to the DRAM 33 via DRAM cache 131. The Read Iterator, e.g 130, and Write Iterators, e.g 132, act as an intermediary between a client, e.g 133, 134, requesting the data and the data stored within the DRAM 33. The iterators are responsible for correct ordering of image data.

Turning to Fig.10, there is a illustrated an example Read Iterator 130 which can comprise a state machine 136 interconnected and controlling a FIFO 137. The state machine 136 is responsible for sending the requests to the DRAM cache and keeping the FIFO 137 full. Further, the state machine 136 receives read requests from clients and clocks-out FIFO data from the FIFO queue 137 in response to those read requests.

The Read Image Iterators 130 can be thought of as a FIFO that contains the entire image in a specific order (of course they are not implemented as such). Every time a pixel is read from the FIFO 137, the next pixel from the image is read into the end of the FIFO.

Write Image Iterators can similarly be considered as a FIFO that is written to by a process. The process writes pixels in a specific order to write out the entire image.

As illustrated in Fig.11, typically a process 140 will have its input tied to a Read Iterator 141, and output tied to a corresponding Write Iterator 142.

A variety of Image Iterators exist to cope with the most common addressing requirements of image processing algorithms. In most cases there is a corresponding Write Iterator for each Read Iterator. The different Iterators are listed in the following table:

| Read Iterators | Write Iterators |
|---|---|
| Sequential Read | Sequential Write |
| Box Read | - |
| Vertical Strip Read | Vertical Strip Write |

In general, more Read Iterators are required than Write Iterators. In the ACP there are 5 Read Iterators and only 3 Write Iterators.

Although an Iterator is perceived to be an unlimited FIFO, in practice there is a small FIFO connected to two or more cache lines. The small FIFO is required to allow for the fact that more than one Iterator is likely to be in use at one time, and only one access can be made to the cache in a single cycle.

All FIFOs belonging to Image Iterators can preferably be accessed by software as memory mapped I/O. General software algorithms that may not be appropriate to be microcoded can therefore take advantage of the image access mechanisms.

Table Access

It can often be necessary to lookup values in a table. Linear table: set up by software eg 256 values of 1 byte each.

ALUs write a byte lookup address to one FIFO,

The linear table address generator looks up the value next cycle (optional multiply by 2 for 16 bit entries) and puts results (8 or 16 bits) into the output FIFO. For 16 bits the order is always same (lo/hi or hi/lo). Value is written to FIFO in cycle N, first 8 bits available from FIFO

at start of N+2 (i.e. skips one cycle).

CCD Image Access

Random Access to pixels

There is no special address generator for specifying fast access to CCD images in DRAM. If a process requires random access it must directly address DRAM and decode image pixels itself.

Sequential Read and Sequential Write Iterators

The simplest Image Iterator is the Sequential Read Iterator. It presents the pixels from a channel one line at a time from top to bottom, and within a line, pixels are presented left to right. The padding bytes are not presented to the client. It is most useful for algorithms that must perform some process on each pixel from an image but don't care about the order of the pixels being processed.

The Sequential Read Iterator comprises 2 cache lines and a small (5 bytes) FIFO. While 32 pixels are being presented from one cache line, the other cache line can be loaded from memory.

Complementing the Sequential Read Iterator is a Sequential Write Iterator. Clients write pixels to a FIFO owned by a Sequential Write Iterator that subsequent writes out a valid image using appropriate caching and appropriate padding bytes. The Sequential Write Iterator again comprises 2 cache lines and a small FIFO.

A process that performs an operation on each pixel of an image independently would typically use a Sequential Read Iterator to obtain pixels, and a Sequential Write Iterator to write the new pixel values to their corresponding locations within the destination image. It is valid to have the source image and destination image to be the same, since a given input pixel is not read more than once.

Internal Format Image Access

Further, as on a single cycle 4 bytes can be transferred from an Iterator's cache into the FIFO, this allows up to 4 Iterators to do the same thing if cache

accesses are staggered. The net effect is that 4 Iterator FIFOs can be accessed every clock cycle without the caches having to support multiple accesses per cycle. 4 Iterators may be 3 Read Iterators and one Write Iterator. For example, as shown in Fig. 12, a single cycle it is possible to read 3 pixels, 1 from each of 3 Read Iterators 145-147, perform some processing on them 148, and take the single pixel output (derived from a previously read 3 pixels) and transfer it to a Write Iterator 149. The average processing time for a single pixel in output would thus be 1 cycle.

A variety of Image Iterators exist to cope with the most common addressing requirements of image processing algorithms. They are:

Sequential Read (previously discussed)

Sequential Write (previously discussed)

Box Read

Vertical Strip Read

Vertical-Strip Write

Box Read Iterator

The Box Read Iterator is used to present pixels in an order most useful for performing general-purpose filters, convolves and the like. The Iterator presents pixel values in a square box around the sequentially read pixels. The box is limited to being 3, 5, or 7 pixels wide. The client has the choice of duplicating edge pixels, or having non-image pixels to be a constant value. The client also has the option of starting the center pixel of

IteratorSpecific$_1$:

The special purpose register IteratorSpecific$_1$ has the following bit usage:

| Bits | Name | Usage |
|---|---|---|
| 0 | DuplicateEdgePixels | 1 = duplicate edge pixels for box region outside image<br>0 = return OutsideImagePixel for box region outside image |

| | | |
|---|---|---|
| 1-8 | OutsideImage Pixel | Constant pixel value to return for pixels outside the actual image area if DuplicateEdgePixels = 0. |
| 9-11 | Reserved | - |

In addition, the special purpose register AGSMSpecific1 is used to determine a sub-sampling in terms of which input pixels will be used as the center of the box. The usual value is 1, which means that each pixel is used as the center of the box. The value "2" would be useful in scaling an image down by 4:1 as in the case of building an image pyramid. Using pixel addresses from the previous diagram, the box would be centered around pixel 0, then 2, 8, and 10.

In Fig.13 there is shown a first example of the box read iterator output with Fig.14 showing a second example. In Fig.13, a box region, e.g 150, is output for a current input pixel 151 with Fig.13 illustrating the 3x3 pixel output case. A first series of pixels 152 illustrates the box read iterator output for the current pixel 151 when duplication of edge pixels is set. A second series of output pixels 153 illustrates the case when duplication of edge pixels is not set. In this case, a pre-set constant "outside image" pixel value is output. Fig.14 illustrates a similar case for the current pixel 156 having a 3x3 output grid 155.

As illustrated in Fig.15, a process that uses the Box Read Iterator 160 for input would most likely use the Sequential Write Iterator 161 for output since they are in sync. A good example is the convolver 162, where N input pixels are read to calculate 1 output pixel.

The Box Read Iterator will require a maximum of 14 (2 x 7) cache lines and a small (5 bytes) FIFO. While pixels are presented from one set of cache lines, the other cache lines can be loaded from memory.

Vertical-Strip Read and Write Iterators

In some instances it is necessary to write an image in

output pixel order, with no knowledge about the direction of coherence in input pixels in relation to output pixels. Examples of this are rotation and warping. If it is necessary to rotate an image 90 degrees, and process the output pixels horizontally, a complete loss of cache coherence may result. On the other hand, if it is necessary to process the output image one cache line's width of pixels at a time and then advance to the next line (rather than advance to the next cache-line's worth of pixels on the same line), we will gain cache coherence for some input image pixels.

It can also be the case that there is known 'block' coherence in the input pixels (such as colour coherence), in which case the read governs the processing order, and the write, to be synchronised, must follow the same pixel order.

With the vertical strip Iterators, the order of pixels presented as input (Vertical-Strip Read), or expected for output (Vertical-Strip Write) is the same and is depicted in Fig. 16. The order is pixels 0 to 31 (165) from line 0 (166), then pixels 0 to 31 of line 1 (167) etc., for all lines of the image, thereby making up first strip 169, then pixels 32 to 63 of line 0, pixels 32 to 63 of line 1 etc., making up second strip 170. In the final vertical strip there may not be exactly 32 pixels wide. In this case only the actual pixels in the image are presented or expected as input.

Referring to Fig.17, a process 173 that requires only a Vertical-Strip Write Iterator will typically have a way of mapping input pixel coordinates given an output pixel coordinate. It would access 175 the input image pixels according to this mapping, and coherence is determined by having sufficient cache lines on the 'random-access' reader for the input image.

It is not meaningful to pair this Write Iterator with a Sequential Read Iterator or a Box read Iterator, but a Vertical-Strip Write Iterator does give significant

improvements in performance in certain situations.

Clients read pixels from the FIFO owned by the Vertical-Strip Read Iterator that reads images cached appropriately. Clients write pixels to the FIFO owned by the Vertical-Strip Write Iterator that subsequent writes out a valid image using appropriate caching and appropriate padding bytes. Each Iterators requires 2 cache lines, and a small (5 byte) FIFO.

Table I/O Units

It is often necessary to lookup values in a table (which may also be an image). While Image Iterators only have a single FIFO, Table I/O Units require 2 FIFOs - an input FIFO and an output FIFO. Clients pass indexes into the Input FIFO (17 bits wide) and receive values from the table via the Output FIFO (16 bits wide).

1 Dimensional Tables

Direct Lookup

A direct lookup is a simple indexing into a 1 dimensional lookup table. The value passed in by the client via the Input FIFO is shifted to the appropriate location using a Barrel Shifter, ANDed with a mask, and then ORed with the Base Address to give the final address. The 8 or 16 bit data value at the address is placed into the Output FIFO. Address generation takes 1 cycle, and transferring the requested data from the cache to the Output FIFO also takes 1 cycle (assuming a cache hit).

Interpolate table

This is the same as a linear table except that 2 values are returned for a given address: The value returned are Table[X], and Table[X+1]. If X+1 is invalid, Table[X] is returned twice. Address generation takes 1 cycle, and transferring the requested data from the cache to the Output FIFO takes 2 cycles (assuming a cache hit).

DRAM FIFO

A special case of a 1D table is a DRAM FIFO. It is often necessary to have a simulated FIFO of a given length

using DRAM and associated caches. With a DRAM FIFO, clients do not index explicitly into the table, but read and write to the table as if it were a large FIFO. Two 2 counters keep track of input and output positions in the simulated FIFO, and cache to DRAM as needed. When values are taken from the Output FIFO by the client, the next values are placed into the FIFO from the cache. When values are placed into the Input FIFO by the client, they are placed into the cache at the next position.

2 Dimensional Tables

Direct Lookup

A 2 dimensional direct lookup is not included at the moment. All cases of 2D lookups are needed for bi-linear interpolation.

Bi-Linear lookup

This kind of lookup is necessary for bi-linear interpolation. Given an X and Y coordinate in a table 4 values are returned after lookup. The four values (in order) are:

Table[X, Y]
Table[X+1, Y]
Table[X, Y+1]
Table[X+1, Y+1]

The order specified allows for the best cache coherence.

3 Dimensional Lookup

Direct Lookup

A 3 dimensional direct lookup is not required at the moment. All cases of 3D lookups are needed for tri-linear interpolation.

Tri-linear lookup

This kind of lookup is necessary for tri-linear interpolation. Given an X, Y, and Z coordinate, 8 values are returned in order from the lookup table:

Table[X, Y, Z]
Table[X+1, Y, Z]
Table[X, Y+1, Z]

Table[X+1, Y+1, Z]
Table[X, Y, Z+1]
Table[X+1, Y, Z+1]
Table[X, Y+1, Z+1]
Table[X+1, Y+1, Z+1]

The 3 values passed in by the client are barrel shifted, ORed together with the base address, and looked up. The 8 sets of 1 byte values are returned via the Output FIFO.Image Pyramid Access

During brushing, tiling, and warping it is often necessary to compute the average colour of a particular area in an image. Rather than calculate the value for each area given, these functions make use of an image pyramid as previously illustrated in Fig. 7. An image pyramid is effectively a multi-resolution pixel-map. The original image is a 1:1 representation. Low-pass filtering and sub-sampling by 2:1 in each dimension produces an image ¼ the original size. This process continues until the entire image is represented by a single pixel.

To access an image pyramid a list of image level addresses is required. These are 12 x 32 bit registers, each stores the address of a given level in the pyramid in the RDRAM memory. The width and height of the original image (level 0) is also required.

The client specifies a pixel address in terms of 3 components: x, y, and level. On subsequent cycles, 4 pixel units are returned in a specific order via a FIFO:
The pixel at (INTEGER[scaled x], INTEGER[scaled y], z)
The pixel at (INTEGER[scaled x]+1, INTEGER[scaled y], z)
The pixel at (INTEGER[scaled x], INTEGER[scaled y]+1, z)
The pixel at (INTEGER[scaled x]+1, INTEGER[scaled y]+1, z)

The offset from the start of an image to a given (x, y) coordinate is given by: RowBytes * Y + X.
For a different level of the pyramid, a simple barrel shift right of the RowBytes value by the level number gives the

RowBytes value for that level. This value needs to be multiplied by a scaled Y (also barrel shifted) and the result added to a barrel shifted X value. For example, if the scaled (X, Y) coordinate was (10.4, 12.7) 4 pixels would be returned in the order (10, 12), (11, 12), (10, 13) and (11, 13). When pixels are exactly aligned (no fractional component), the "+1" pixels are duplicated (to save a read from DRAM). When a coordinate is outside the valid range, clients have the choice of edge pixel duplication or returning of a constant colour value (typically black).

DRAM Interface 81

The DRAM used by the Artcam is a 64Mbit (8MB) RAMBUS Dram operating at 500MHz. Using RAMBUS DRAM implies that applications should minimize the number of random memory accesses to avoid degraded memory access performance.

To take advantage of the 4 internal banks of memory in a single DRAM chip, every 32 bytes should be in a different bank with address wiring accordingly. The 4 bank internal arrangement of RAMBUS DRAM can also be used to advantage if necessary as long as this does not create unnecessary algorithmic complexity.

Bank accesses can have their latencies overlapped, so while data is being transferred from one bank, another can be setting up for the transfer. Interleaved in this way, assuming a worst case of a DRAM-internal-cache miss every access, 4 sets of 32 byte reads can be accomplished in 320ns.

Cache Lines

In order to reduce effective memory latency, the ACP contains 128 cache lines, each 32 bytes wide. The total memory on chip for caches is therefore 4096 bytes (128 x 32 bytes). The breakup of cache assignment is:

16 to cache the CPU's program (so programs can run at the same time as control ACP processes)

16 to cache CPU program's data

96 floating. These can be assigned to ALUs for

particular functions, or assigned to CPU program or data as desired.

The 128 cache lines are divided into 8 groups of 16 for separate addressing in a given cycle, with appropriate multiplexing.

Memory Organization

Memory in an Artcam consists of a contiguous 32MB area (of which 8 MB is actually used). In addition to the *real* memory, there are some other non-contiguous address spaces which are effectively 'virtual' memory areas. These are ACP registers, used for memory mapped I/O. The memory organization for an Artcam with 8MB of RDRAM is shown in the following table:

| | |
|---|---|
| Program scratch RAM | 0.50 MB |
| Artcard data | 1.00 MB |
| Photo Image, captured from CCD | 0.50 MB |
| Print Image (compressed) | 2.25 MB |
| 1 Channel of expanded Photo Image | 1.50 MB |
| 1 Image Pyramid of single channel | 1.00 MB |
| Intermediate Image Processing | 1.25 MB |
| **TOTAL** | **8 MB** |

Uncompressed, the Print Image requires 4.5MB (1.5MB per channel). To accommodate other objects in the 8MB model, the Print Image needs to be compressed. If the chrominance channels are compressed by 4:1 they require only 0.375MB each). The memory model described here assumes a single 8 MB RDRAM. Other models of the Artcam may have more memory, and thus not require compression of the Print Image. In addition, with more memory a larger part of the final image can be worked on at once, potentially giving a speed improvement. The ejecting or inserting an Artcard invalidates the 5.5MB area holding the Print Image, 1 channel of expanded photo image, and the image pyramid. This space may be safely used by the Artcard Interface for

decoding the Artcard data.

VLIW Vector Processor 74

In order to reduce the complexity of the ACP design, the ACP contains a VLIW (Very Long Instruction Word) Vector Processor 74. The processor is essentially a set of I/O Units 177 connected to a set of ALUs 178 via FIFOs 179 as illustrated in Fig. 18. The Cache Interface 176 is described separately below. It provides the interface to DRAM 33 and is the primary input and output mechanism for the VLIW 74.

The I/O Units Block 177 consists of a number of types of address generators, each linked to a specific FIFO and the Cache Interface. The address generators are able to read and write data (specifically images in a variety of formats) as well as tables and simulated FIFOs in DRAM. They are customizable under software control, but cannot be microcoded.

The FIFOs 179 connecting the I/O Units 177 to the ALUs 178 are tied to specific I/O Units and specific ALUs. In summary there are:

5 x 8 bit output FIFOs (from I/O unit to ALU)

3 x 8 bit input FIFOs (from ALU to I/O unit)

4 x 16 bit output FIFOs (from I/O unit to ALU)

4 x 17 bit input FIFOs (from ALU to I/O unit)

External processes have the ability to write to 1 of these 8 bit input FIFOs, and to read from 1 of the 8 bit output FIFOs. This allows other parts of the chip to provide input (for example the Image Sensor Interface can provide the pixels from the CCD) or to process the output (for example the Print Head Interface is able to take pixels in order to print them). These two FIFOs are known as the VLIW Input FIFO 180 and VLIW Output FIFO 181 respectively.

The ALUs Block 178 consists of a number of types of microprogrammable ALUs coupled together. Each of the ALUs contains a number of registers, some microcode RAM, and connections to the outside world. The connections are

inputs, outputs, or both inputs and outputs. Specific ALUs connect to the FIFOs 179 and via them to the Address Units.

The Address and Data Buses connection 182 allows the CPU to read and write registers in the VLIW Vector Processor, as well as each ALU's microcode RAM. Rather than have the microcode in ROM inside the VLIW Vector Processor, the microcode is in RAM, with the program CPU responsible for loading it up. For the same space on chip, this tradeoff reduces the maximum size of any one function to the size of the RAM, but allows an unlimited number of functions to be written in microcode. Functions implemented using ALU microcode include Vark acceleration, Artcard reading, and Printing functions.

The VLIW Vector Processor scheme has several advantages for the case of the ACP:

Hardware design complexity is reduced

Hardware risk is reduced due to reduction in complexity

Hardware design time does not depend on all Vark functionality being implemented in dedicated silicon.

Space on chip is reduced overall (due to large number of processes able to be implemented as microcode)

Functionality can be added to Vark (via microcode) with no impact on hardware design time.

ALUs Block 178

The ALUs Block 178 consists of a number of types of microprogrammable ALUs coupled together. Each of the ALUs contains a number of registers, some microcode RAM, and connections to the outside world. The connections are inputs, outputs, or both inputs and outputs. Specific ALUs connect to the FIFOs and via them to the I/O Units.

The different ALU types are:

Memory Interface Units: connected to the FIFOs

- Read Unit - attached to FIFO corresponding to a Read Iterator

- Write Unit - attached to a FIFO corresponding to a Write Iterator
- ReadWrite Unit - attached to 2 FIFOs corresponding to Table I/O Unit

Processing Units:

- Adder ALU - for counters, comparisons and simple loops.
- Multiply ALU - single cycle multiply/accumulate for interpolations and convolves
- Logical ALU - for bit manipulation

A summary of each type of ALU Unit is listed in the following table:

| ALU Unit Name | # of Registers | # of Data Outputs | # of Status Outputs | # of Control Outputs | Size of Microcode RAM |
|---|---|---|---|---|---|
| Read | 1 | 1 | - | 1 | 800 bits |
| Write | 1 | - | - | 1 | 704 bits |
| ReadWrite | 2 | 1 | - | 1 | 1216 bits |
| Adder | 4 | 3 | 1 | - | 1632 bits |
| Multiply | 4 | 4 | 2 | - | 1920 bits |
| Logical | 4 | 2 | 1 | - | 1376 bits |

The outputs from the units are connected to the inputs so that each unit can select input from both its own outputs and all other units' outputs. The structure is as illustrated in Fig. 143. As shown in Fig. 143, there are multiple copies of each unit. The following table lists how many of each type of unit are present, and provides an overall total of specific resources.

| Unit Name | # of Units | # of Registers | # of Data Outputs | # of Status Outputs | # of Control Outputs | Size of Microcode RAM |
|---|---|---|---|---|---|---|
| Read | 5 | 5 | 5 | - | 1 | 4000 bits |
| Write | 3 | 3 | - | - | 1 | 2112 bits |
| ReadWrite | 4 | 8 | 4 | - | 1 | 4864 bits |
| Adder | 4 | 16 | 12 | 4 | - | 6528 bits |
| Multiply | 4 | 16 | 16 | 4 | - | 7680 bits |
| Logical | 2 | 8 | 4 | 2 | - | 2752 bits |
| TOTAL | 24 | 38 | 41 | 10 | 1 | 27936 bits |

All Units connected to FIFOs produce a control bit. The control bits are ORed together to produce the SuspendALUs control bit, which is passed as input into **every** unit. The bit will be set if an attempt is due to be made this cycle to access a FIFO which is not ready (e.g. it is being written to and it is full). If set, all ALUs are suspended for the cycle, and no processing takes place. Processing will be suspended until the SuspendALUs control bit is clear (e.g. if the FIFO is now ready). This mechanism is provided so that synchronization is not an issue. While this does not provide optimum performance, it does considerably reduce hardware and software (microcode) design complexity.

The total number of data outputs is 41. This implies 6 bits are necessary in order to select 1 input from the available outputs.

The total number of status outputs is 10. Since each status output consists of 2 bits (a N (Negative) bit and a Z (Zero) bit), there are actually 20 status bits. Consequently 5 bits are necessary in order to select 1 from the 20 status bits.

Memory Interface Units

In order to transfer data between the various ALUs and the memory, a variety of units have been introduced. They include Read, Write, and ReadWrite units.

In order to reduce complexity of microcode, all units hang if any one of them requires memory access and the FIFO is not yet available (for reading or for writing). This mechanism is provided by the SuspendALUs control bit, described in Note$_1$ of the previous section.

The memory interface units do not access DRAM nor the caches themselves. They merely provide an interface between other ALUs and the memory, providing a timing and synchronization buffer via the FIFOs.

Read Unit

The Read Unit provides data from DRAM. Specifically, it is attached to a FIFO that is filled by a Read Iterator. The Read Unit is attached to the output end of this FIFO, and does not concern itself with how data is inserted into the FIFO.

The Read Unit structure is set out in Fig. 143 and 1 data output and no status outputs, although if a read is requested from the FIFO, and the FIFO is empty, then the entire ALU microcode is disabled until the FIFO has something inside.

Microcode RAM

The Microcode RAM for the Read Unit is a 32 entry by 25 bit RAM (800 bits), containing the program for the ALU. The meaning of each of the microcode control bits is described here:

| Bits | # Bits | Description |
|---|---|---|

| | | |
|---|---|---|
| 0 | 1 | Read from FIFO |
| 1 | 1 | Sign Extend$_1$ to 32 bits (input to BarrelShift$_1$)<br>0 = no sign extend (pad with 0's)<br>1 = sign extend |
| 2-3 | 2 | BarrelShift$_1$ (shifts left only, padding lower bits with 0)<br>00 = no shift<br>01 = shift left 8 bits<br>10 = shift left 16 bits<br>11 = shift left 24 bits |
| 4-7 | 4 | Write Enable to Latch (each enable-bit represents 1 byte) |
| 8 | 1 | Sign Extend$_2$ (input to Bit Fiddler)<br>0 = no sign extend (pad with 0's)<br>1 = sign extend |
| 9-11 | 3 | Bit Fiddler (Generates 32 bit number from 32 bit number ABCD)<br>000 = XXXA<br>001 = XXXB<br>010 = XXXC<br>011 = XXXD<br>100 = XXAB<br>101 = XXBC<br>110 = XXCD<br>111 = ABCD |
| 12-13 | 2 | BarrelShift$_2$ (shifts left only, padding lower bits with 0)<br>00 = no shift<br>01 = shift left 8 bits<br>10 = shift left 16 bits<br>11 = shift left 24 bits |
| 14-18 | 5 | Select input status bit to compare against (branch if equal)<br>00000 - 11101 = select input status bit |

| | | |
|---|---|---|
| | | 11110 = don't jump (address is next microcode word)<br>11111 = always jump (regardless of status) |
| 19 | 1 | Value to compare status bit against (branch if matches) |
| 20-24 | 5 | Address to jump to (if branching) |
| | 25 | TOTAL |

Write Unit

The Write Unit is illustrated in Fig. 145 and provides the interface of writing to DRAM to the ALU programs. Specifically, it is attached to a FIFO that is read/emptied by a Write Iterator. The Write Unit is attached to the input end of this FIFO, and does not concern itself with how data is removed from the FIFO.

The Write Unit does not output data to any other ALUs, although if a write is requested from the FIFO, and the FIFO is full, then the SuspendALUs signal is generated until the FIFO can be written to.

Microcode RAM

The Microcode RAM is a 32 entry by 22 bit RAM (704 bits), containing the program for the ALU. The meaning of each of the microcode control bits is described here:

| Bits | # Bits | Description |
|---|---|---|
| 0-5 | 6 | Select input from other units |
| 6 | 1 | Write Enable to Latch |
| 7 | 1 | Select $IN_1$ or data from Latch |
| 8-9 | 2 | 8 bit select from 32 bits (ABCD)<br>00 = D<br>01 = C<br>10 = B |

| | | |
|---|---|---|
| | | 11 = A |
| 10 | 1 | Write to FIFO |
| 11-15 | 5 | Select input status bit to compare against (branch if equal)<br>00000 - 11101 = select input status bit<br>11110 = don't jump (address is next microcode word)<br>11111 = always jump (regardless of status) |
| 16 | 1 | Value to compare status bit against (branch if matches) |
| 17-21 | 5 | Address to jump to (if branching) |
| | 22 | TOTAL |

ReadWrite Unit

The ReadWrite Unit is illustrated in Fig. 146 and provides mechanisms for reading (and writing) into lookup tables and creating DRAM FIFOs. The ReadWrite Unit has both input and output, and attaches to 2 FIFOs that are in turn connected to address generators that can interpret requests for lookup data. Note that in a single cycle,

Clients send their requests to the ReadWrite Unit, which in turn passes their requests into a FIFO. Results from the request (in the case of a Read request) are then read from the second FIFO. Note that these two FIFOs are not the same as the 8 bit FIFOs attached to the Read and Write Units. Instead there is a 17 bit output FIFO (1 bit for request, 16 for data), and a 16 bit input FIFO.

Microcode RAM

The Microcode RAM is a 32 entry by 38 bit RAM (1216 bits), containing the program for the ALU. The meaning of each of the microcode control bits is described here:

| Bits | # Bit | Description |
|---|---|---|

| | s | |
|---|---|---|
| 0-5 | 6 | Select input from other units |
| 6 | 1 | Write Enable to $Latch_2$ |
| 7 | 1 | Select $IN_1$ or data from $Latch_2$ |
| 8-9 | 3 | 16 bit select from 32 bits (ABCD)<br>000 = 0D<br>001 = 0C<br>010 = 0B<br>011 = 0A<br>100 = CD<br>101 = BC<br>110 = AB<br>111 = 0 |
| 10 | 1 | Write to FIFO |
| 11 | 1 | Request Bit to send as $17^{th}$ bit in input to FIFO |
| 12 | 1 | Read from FIFO |
| 13-14 | 2 | Sign $Extend_1$ to 32 bits (input to $BarrelShift_1$)<br>00 = no sign extend (pad with 0's)<br>01 = sign extend using bit 7<br>10 = sign extend using bit 15<br>11 = reserved |
| 15-16 | 2 | $BarrelShift_1$ (shifts left only, padding lower bits with 0)<br>00 = no shift<br>01 = shift left 8 bits<br>10 = shift left 16 bits<br>11 = shift left 24 bits |
| 17-20 | 4 | Write Enable to Latch (each enable-bit represents 1 byte) |
| 21 | 1 | Sign $Extend_2$ (input to Bit Fiddler)<br>0 = no sign extend (pad with 0's)<br>1 = sign extend |
| 22-24 | 3 | Bit Fiddler (Generates 32 bit number |

| | | |
|---|---|---|
| | | from 32 bit number ABCD)<br>000 = XXXA<br>001 = XXXB<br>010 = XXXC<br>011 = XXXD<br>100 = XXAB<br>101 = XXBC<br>110 = XXCD<br>111 = ABCD |
| 25-26 | 2 | $BarrelShift_2$ (shifts left only, padding lower bits with 0)<br>00 = no shift<br>01 = shift left 8 bits<br>10 = shift left 16 bits<br>11 = shift left 24 bits |
| 27-31 | 5 | Select input status bit to compare against (branch if equal)<br>00000 - 11101 = select input status bit<br>11110 = don't jump (address is next microcode word)<br>11111 = always jump (regardless of status) |
| 32 | 1 | Value to compare status bit against (branch if matches) |
| 33-37 | 5 | Address to jump to (if branching) |
| | 38 | TOTAL |

Processing Units

Adder ALU

As illustrated in Fig. 147, each adder ALU is a simple 32 bit adder with Min and Max functionality, a barrel shifter, and 4 registers.3 sets of 32 bit values as well as Negative and Zero status bits are provided as outputs from the ALU. In addition, each ALU has a microcode RAM containing small programs with limited branching ability.

The Adder ALU is designed to perform addition, simple averaging (e.g. add 2 numbers and divide by 2), and provide mechanisms for looping and control for other ALUs (via status bits).

Microcode RAM

The Microcode RAM is a 32 entry by 51 bit RAM (1632 bits), containing the program for the ALU. The meaning of each of the microcode control bits is described here:

| Bits | # Bits | Description |
|---|---|---|
| 0-5 | 6 | Select $IN_1$ from outputs from this and other ALUs |
| 6-11 | 6 | Select $IN_2$ from outputs from this and other ALUs |
| 12-17 | 6 | Select $IN_3$ from outputs from this and other ALUs |
| 18-19 | 2 | Select $OUT_1$ from 4 registers |
| 20-21 | 2 | Select $OUT_2$ from 4 registers |
| 22-23 | 2 | Select register to write to [from 4 registers] |
| 24 | 1 | WriteEnable to register<br>0 = don't write<br>1 = write to specified register |
| 25 | 1 | Select $AdderInput_1$ from [0, $IN_2$] |
| 26 | 1 | Select RegisterInput from [0, $IN_1$] |
| 27 | 1 | Negate $AdderInput_1$ |
| 28-29 | 2 | Select Function [MIN, MAX, +, ABS(+)] |
| 30-31 | 2 | Operation resolution [input to MIN, MAX, +, TST, ABS]<br>00 = 32 bits<br>01 = 16 bits<br>11 = 8 bits |
| 32 | 1 | Limit to 0 min [input to MIN, MAX, |

| | | |
|---|---|---|
| | | +] |
| 33 | 1 | Treat as signed [input to MAX, MIN, +, and Barrel Shifter] |
| 34 | 1 | $C_{in}$ [input to +] |
| 35 | 1 | WrapEnable[input to +]<br>If set, addition is allowed to wrap. If clear, addition will ceiling and floor at appropriate value for the resolution and its signed/unsigned nature. |
| 36 | 1 | Direction for barrel shift [sign extended if signed]<br>0 = left<br>1 = right |
| 37-39 | 3 | #Bits to shift [input to Barrel Shifter]<br>000 = 0<br>001 = 1<br>010 = 2<br>011 = 3<br>100 = 4<br>101 = 5<br>110 = 8<br>111 = 16 |
| 40-44 | 5 | Select input status bit to compare against (branch if equal)<br>00000 - 11101 = select input status bit<br>11110 = don't jump (address is next microcode word)<br>11111 = always jump (regardless of status) |
| 45 | 1 | Value to compare status bit against (branch if matches) |
| 46-50 | 5 | Address to jump to (if branching) |

| | | |
|---|---|---|
| | 51 | TOTAL |

Multiply ALU

As illustrated in Fig. 148, each multiply ALU is a 32 bit multiply/accumulator. It is designed for high speed interpolation and convolving, and includes a barrel shift on output for user-specified precision. The Multiply ALU therefore has 4 data outputs, and 2 status outputs.

Microcode RAM

The Microcode RAM is a 32 entry by 60 bit RAM (1920 bits), containing the program for the ALU. The meaning of each of the microcode control bits is described here:

| Bits | # Bits | Description |
|---|---|---|
| 0-5 | 6 | Select $IN_1$ from outputs from this and other ALUs |
| 6-11 | 6 | Select $IN_2$ from outputs from this and other ALUs |
| 12-17 | 6 | Select $IN_3$ from outputs from this and other ALUs |
| 18-23 | 6 | Select $IN_4$ from outputs from this and other ALUs |
| 24-25 | 2 | Select $OUT_1$ from 4 registers |
| 26-27 | 2 | Select $OUT_2$ from 4 registers |
| 28-29 | 2 | Select register to write to [from 4 registers] |
| 30 | 1 | WriteEnable to register<br>0 = don't write<br>1 = write to specified register |
| 31 | 1 | Select $AdderInput_1$ from [0, $IN_2$] |
| 32 | 1 | Negate $AdderInput_1$ |
| 33 | 1 | Select RegisterInput from [0, $IN_1$] |
| 34-35 | 2 | Select 16 bits from $In_3$<br>00 = Low 8 bits (pads high 8 bits with 0) |

| | | |
|---|---|---|
| | | 01 = Low 16 bits<br>10 = Mid 16 bits<br>11 = High 16 bits |
| 36-37 | 2 | Select 16 bits from $In_4$ (see above for bit description) |
| 38 | 1 | $BitsToNegate_1$[calculates 1-X]<br>0 = Negate low 8 bits only<br>1 = Negate all 16 bits |
| 39 | 1 | Select between $MultiplyInput_1$ and 1-$MultiplyInput_1$ |
| 40 | 1 | Treat as Signed [input to +, *, and Barrel Shifter]<br>0 = Signed * and +<br>1 = Unsigned * and + |
| 41 | 1 | Operation resolution [input only as output always 32 bits]<br>0 = 16 bits<br>1 = 8 bits |
| 42 | 1 | $C_{in}$ [input to +] |
| 43 | 1 | Limit to 0 min [input to +] |
| 44 | 1 | WrapEnable[input to +]<br>If set, addition is allowed to wrap.<br>If clear, addition will ceiling and floor at appropriate value for the resolution and its signed/unsigned nature. |
| 45 | 1 | Direction for barrel shift [sign extended if signed]<br>0 = left<br>1 = right |
| 46-48 | 3 | #Bits to shift [input to Barrel Shifter]<br>000 = 0<br>001 = 1<br>010 = 2<br>011 = 3<br>100 = 4 |

| | | |
|---|---|---|
| | | 101 = 5<br>110 = 8<br>111 = 16 |
| 49-53 | 5 | Select input status bit to compare against (branch if equal)<br>00000 - 11101 = select input status bit<br>11110 = don't jump (address is next microcode word)<br>11111 = always jump (regardless of status) |
| 54 | 1 | Value to compare status bit against (branch if matches) |
| 55-59 | 5 | Address to jump to (if branching) |
| | 60 | TOTAL |

Logical ALU

As illustrated in Fig. 149, the Logical ALU allows simple logical operations such as AND, OR and XOR functions to be performed. It is specifically useful for preparing operands for interpolation, for merging separately created components of a number, and for bit testing in order to provide control to other units. Take for example, the case of interpolation via lookup. Given an 8 bit number, the lookup may only use 4 bits, and leave the remaining 4 bits to provide the interpolation. The Logical ALU allows the remaining 4 bits to be isolated. The Logical ALU therefore has 2 data outputs and 1 status output.

Microcode RAM

The Microcode RAM is a 32 entry by 43 bit RAM (1376 bits), containing the program for the ALU. The meaning of each of the microcode control bits are described here:

| Bits | # Bits | Description |
|---|---|---|
| 0-5 | 6 | Select $IN_i$ from outputs from this and other |

| | | |
|---|---|---|
| | | ALUs |
| 6-11 | 6 | Select $IN_2$ from outputs from this and other ALUs |
| 12-17 | 6 | Select $IN_3$ from outputs from this and other ALUs |
| 18-19 | 2 | Select $OUT_1$ from 4 registers |
| 20-21 | 2 | Select register to write to [from 4 registers] |
| 22 | 1 | WriteEnable to register<br>0 = don't write<br>1 = write to specified register |
| 23 | 1 | Negate $IN_1$ |
| 24 | 1 | Negate $IN_2$ |
| 25-26 | 2 | Select Logical Function<br>00 = NOT($IN_1$)<br>01 = $IN_1$ AND $IN_2$<br>10 = $IN_1$ OR $IN_2$<br>11 = $IN_1$ XOR $IN_2$ |
| 27 | 1 | Direction for barrel shift<br>0 = left<br>1 = right |
| 28 | 1 | SignExtend [input to Barrel Shifter]<br>0 = no sign extend<br>1 = sign extend when shifting right |
| 29-31 | 3 | #Bits to shift [input to Barrel Shifter]<br>000 = 0<br>001 = 1<br>010 = 2<br>011 = 3<br>100 = 4<br>101 = 5<br>110 = 8<br>111 = 16 |
| 32-36 | 5 | Select input status bit to compare against (branch if equal)<br>00000 - 11101 = select input status bit |

| | | |
|---|---|---|
| | | 11110 = don't jump (address is next microcode word)<br>11111 = always jump (regardless of status) |
| 37 | 1 | Value to compare status bit against (branch if matches) |
| 38-42 | 5 | Address to jump to (if branching) |
| | 43 | TOTAL |

I/O Units 177

The I/O Units Block 177 is illustrated in further detail in Fig. 19 and consists of a number of types of address generators, each linked to a specific FIFO and the Cache Interface. The address generators are able to read and write data (specifically images in a variety of formats) as well as tables and simulated FIFOs in DRAM. They are customizable under software control, but cannot be microcoded.

The types of address generators are:

Read Image Iterators 190, used to iterate through pixels of an image in a variety of ways

Write Image Iterators 191, used to write pixels of an image in a variety of ways, and

Table I/O Units 192, used to randomly access pixels in images, data in tables, and to simulate FIFOs.

There are a total of:

5 Read Image Iterators 190, each connected to an 8 bit output FIFO

3 Write Image Iterators 191, each connected to an 8 bit input FIFO

4 Table I/O Units 192, each connected to a 16 bit output FIFO a 17 bit input FIFO

Each of the address generators is connected to one of the 7 Cache Interface ports (the 8$^{th}$ is reserved for the CPU). all FIFOs can be accessed by software as memory mapped I/O.

Interpolation using ALUs

Interpolation is heavily used in image creation by the ACP, from simple compositing through to tri-linear interpolation for colour space conversion. Interpolation is defined in one of two forms, with the value at fractional position f between A and B given by: A + (B-A)f, or as A(1-f) + fB.

Both forms reduce to the same implementation. Rather than have specific interpolation hardware, it is possible to microcode interpolation using the ALUs. Interpolation can be implemented in a variety of ways using different numbers of ALUs depending on the other functions required at the same time. The following is a sample of interpolation methods in the general sense only. The method of interpolation & hence number of ALUs required etc. is described as required for each use of interpolation within the ACP.

Both forms can be reduced to the same implementation. Rather than have specific interpolation hardware, it is possible to microcode interpolation using the ALUs. It is therefore possible to set up a single Adder ALU and Multiply ALU to work in conjunction so that they effectively form a pipeline that produces the result of a single interpolation every clock cycle after a 2 cycle setup delay). Sample microcode pseudocode for interpolation of a 1 dimensional data stream (given by Invalue) is:

| Cycle | Multiply ALU | Adder ALU |
|---|---|---|
| 1 | A = Invalue | A = 0 |
| 2 | Mult.Out1 = A<br>Calculate f * Adder.Out1 + Mult.Out1<br>B = Invalue | Out1 = A,<br>B = Invalue - Mult.Out1 |
| 3 | Mult.Out1 = B<br>Calculate f * Adder.Out1 + Mult.Out1 | A = Invalue<br>Goto 2 |

| | | |
|---|---|---|
| | Goto 2 | |

It is also possible to perform interpolation on data coming in as pairs of values in a single input stream. In this case we only need 1 Multiply ALU, there is a pipeline delay of 2 cycles, and the process takes 2 cycles on average.

| Cycle | Multiply ALU |
|---|---|
| 1 | Acc = (1-f) * Invalue |
| 2 | Mult.Out1 = A<br>Acc = Acc + (f * Invalue) |
| 3 | Mult.Out = Acc<br>Acc = (1-f) * Invalue<br>Goto 2 |

Pairs of data in 2 streams

If data is coming in as pairs of values from 2 input streams, we can get by with 1 Multiply ALU and 1 Adder ALU. In this case we can interpolate in 1 cycle on average.

| Cycle | Multiply ALU | Adder ALU |
|---|---|---|
| 1 | A = Invalue | A = Invalue1 - Invalue2 |
| 2 | Mult.Out1 = A<br>Calculate f * Adder.Out1 + Mult.Out1<br>B = Invalue | Out1 = A,<br>B = Invalue1 - Invalue2 |
| 3 | Mult.Out1 = B<br>Calculate f * Adder.Out1 + Mult.Out1<br>A = Invalue<br>Goto 2 | Out1 = B,<br>A = Invalue1 - Invalue2<br>Goto 2 |

Bi-linear interpolation

In bi-linear interpolation a total of 3 interpolations need to be performed:

2 interpolations between the 2 pairs of data

1 interpolation between the output of the 2 interpolations

If the data is coming in from a single stream, we can choose for optimizing for speed or ALU usage. If we wish to minimize ALU usage, we can perform 1 interpolation per 2 cycles using a single Multiply ALU. Thus the time required for the 3 interpolations is 6 cycles. Alternatively we can use 2 Multiply ALUs: perform the 2 interpolations in 4 cycles using 1 Multiply ALU, and perform the remaining interpolation in 2 cycles with the other Multiply ALU. Since the 2 Multiply ALUs work in parallel, the total time for tri-linear interpolation would be 4 cycles.

If the data is coming in from 2 streams, we can again optimize for speed or ALU usage. If we wish to minimize ALU usage, we can perform 1 interpolation per 2 cycles using a single Multiply ALU. Thus the time required for the 3 interpolations in a bi-linear interpolation is 6 cycles. We can also use 1 Multiply ALU with an Adder ALU (see Pairs of data in 2 streams, detailed above), giving 1 interpolation per cycle (on average) and hence 3 cycles for the bi-linear interpolation. Alternatively we can use 3 Multiply ALUs in combination with 3 Adder ALUs to give an average throughput of 1 cycles.

Tri-linear interpolation

In tri-linear interpolation a total of 7 interpolations need to be performed:

4 interpolations, 1 between each of the 4 pairs of data

2 interpolations between the output of the 4 interpolations

1 interpolation between the output of the 2 interpolations

If the data is coming in a single stream, we can choose between optimizing for speed or ALU usage. If we wish to

minimize ALU usage, we can perform 1 interpolation per 2 cycles using a single Multiply ALU. Thus the time required for the 7 interpolations in a tri-linear interpolation is 14 cycles. Alternatively, we can use 2 Multiply ALUs: perform the 4 interpolations in 8 cycles using 1 Multiply ALU, and perform the remaining 3 interpolations in 6 cycles using the other Multiply ALU. Since the 2 Multiply ALUs work in parallel, the total time for tri-linear interpolation would be 8 cycles.

If the data is coming in 2 streams, it is possible to again optimize for speed or ALU usage. If it is necessary to minimize ALU usage, it is possible to perform 1 interpolation per 2 cycles using a single Multiply ALU. Thus the time required for the 7 interpolations in a tri-linear interpolation is 14 cycles. It is possible to also use 1 Multiply ALU with an Adder, giving 1 interpolation per cycle (on average) and hence 7 cycles for the tri-linear interpolation. Alternatively we can use all 4 Multiply ALUs in combination with all 4 Adder ALUs to give an average throughput of 2 cycles.

Generation of Coordinates using VLIW Vector Processor

Some functions that are linked to Write Iterators require the X and Y coordinates of the current pixel being processed in part of the processing pipeline. Particular processing may need to take place at the end of each row, or column being processed.

Each function requiring coordinates will have a different pixel calculation time, and as such will have slightly different timing for coordinate generation. However, The essence and ALU requirements will be the same in each instance, however.

Generate Sequential [X, Y]

When a process is processing pixels in sequential order according to the Sequential Read Iterator (or generating pixels and writing them out to a Sequential Write Iterator), the process as shown in Fig. 20 can be used to generate X, Y

coordinates. One form of implementation is as shown in Fig. 21. The coordinate generator counts up to ImageWidth in the X ordinate, and once per ImageWidth pixels increments the Y ordinate. The following constants of Fig. 21 are set by software:

| Constant | Value |
|---|---|
| $K_1$ | ImageWidth |
| $K_2$ | ImageHeight (optional) |

The following registers are used to hold temporary variables:

| Variable | Value |
|---|---|
| $Latch_1$ | X (starts at 0 each line) |
| $Latch_2$ | Y (starts at 0) |

The requirements are summarized as follows:

| Requirements | *+ | + | K | LU | Iterators |
|---|---|---|---|---|---|
| General | 0 | 3/4 | 3/4 | 0 | 0 |
| TOTAL | 0 | 3/4 | 3/4 | 0 | 0 |

Generate Vertical Strip [X, Y]

The vertical strip generation process is as shown in Fig. 22 The coordinate generator simply counts up to ImageWidth in the X ordinate, and once per ImageWidth pixels increments the Y ordinate. An actual implementation is as illustrated in Fig. 23, where the following constants are set by software:

| Constant | Value |
|---|---|
| $K_1$ | 32 |
| $K_2$ | ImageWidth |
| $K_3$ | ImageHeight |

The following registers are used to hold temporary variables:

| Variable | Value |
|---|---|
| $Latch_1$ | StartX (starts at 0, and is incremented by 32 |

| | |
|---|---|
| | once per vertical strip) |
| $Latch_2$ | X |
| $Latch_3$ | EndX (starts at 32 and is incremented by 32 to a maximum of ImageWidth) once per vertical strip) |
| $Latch_4$ | Y |

The requirements are summarized as follows:

| Requirements | *+ | + | K | LU | Iterators |
|---|---|---|---|---|---|
| General | 0 | 4 | 7 | 0 | 0 |
| TOTAL | 0 | 4 | 7 | 0 | 0 |

CPU Memory Decoder

The CPU Memory Decoder is a simple decoder for satisfying CPU data accesses. The Decoder translates data addresses into DRAM addresses (which then get passed on to the Cache Interface) or into internal ACP register accesses over the internal low speed bus. The CPU Memory Decoder allows for memory mapped I/O of ACP registers. A straightforward way of deciding is to use address bit 24. If bit 24 is clear, the address is in the lower 16 MB range, and hence can be directed to the Cache Interface to be satisfied from DRAM. In most cases the DRAM will only be 8 MB, but we allocate 16 MB to cater for a higher memory model Artcam. If bit 24 is set, the address represents an internal ACP register address. The address is translated into an access over the low speed bus to the requested component in the ACP.

Program Cache

A small cache is required for good performance. This requirement is mostly due to the use of a Rambus DRAM, which can provide high-speed data in bursts, but is inefficient for single byte accesses. 16 dedicated cache lines of 32 bytes each will achieve most of the performance gain over no cache, and limits the cache size to 512 bytes. The program cache gives increased performance for the CPU, and even

allows small CPU functions to run completely from cache (and therefore simultaneously with VLIW processes). The Program Cache is a **read only** cache, taking its data from the DRAM Memory Interface. The data used by CPU programs comes through the CPU Memory Decoder and if the address is in DRAM, through the general Cache Interface.

Cache Interface

The ACP contains a dedicated CPU instruction cache and a general data cache interface. The CPU instruction cache is described in the previous chapter, while this chapter discusses the general data cache. In order to reduce effective memory latency, the ACP contains 128 cache lines. Each cache line is 32 bytes wide. Thus the total amount of data cache is 4096 bytes (4k). Each cache line has a 4 bit group number associated with it, thereby allowing the splitting of the caches into 16 different groups. The caching groups must be contiguous sets of cache lines.

All processor data requests use cache request group 0, and although the CPU can assign any number of cache lines (except none) to cache group 0, a minimum of 16 cache lines is recommended for good performance.

The other users of the cache interface - namely the Artcard Interface, the Display Controller, and the VLIW Vector Processor must use cache request groups appropriately. The CPU is responsible for ensuring that a correct number of cache lines is assigned to each cache group for a given process. In any given cycle, 4 simultaneous accesses of 32 bits (4 bytes) to the caches are permitted. Each access must be to a separate group of cache lines.

Serial Interfaces

USB serial port interface

This is a standard USB serial port, which is connected to the internal chip low speed bus, thereby allowing the CPU

to control it.

Keyboard interface

This is a standard low-speed serial port, which is connected to the internal chip low speed bus, thereby allowing the CPU to control it. It is designed to be optionally connected to a keyboard to allow simple data input to customize prints.

Authentication chip serial interfaces

These are 2 standard low-speed serial ports, which are connected to the internal chip low speed bus, thereby allowing the CPU to control them..

The reason for having 2 ports is to connect to both the on-camera Authentication chip, and to the print-roll Authentication chip using separate lines. Only using 1 line may make it possible for a clone print-roll manufacturer to design a chip which, instead of generating an authentication code, tricks the camera into using the code generated by the authentication chip in the camera.

Parallel Interface

The parallel interface connects the ACP to individual static electrical signals. The following is a table of connections to the parallel interface:

| Connection | Direction | Pins |
|---|---|---|
| Paper transport stepper motor | Output | 4 |
| Artcard stepper motor | Output | 4 |
| Zoom stepper motor | Output | 4 |
| Guillotine solenoid | Output | 1 |
| Flash trigger | Output | 1 |
| Status LCD segment drivers | Output | 7 |
| Status LCD common drivers | Output | 4 |
| Artcard illumination LED | Output | 1 |
| Artcard status LED (red/green) | Input | 2 |

| | | |
|---|---|---|
| Artcard sensor | Input | 1 |
| Paper pull sensor | Input | 1 |
| Orientation sensor | Input | 2 |
| Buttons | Input | 4 |
| Total | | 36 |

The CPU is able to control each of these connections as memory mapped I/O via the low speed bus.

Display Controller

Principles of Operation

When the "Take" button on an Artcam is half depressed, the TFT will display the current image from the image sensor (converted via a simple VLIW process). Once the Take button is fully depressed, the Taken Image is displayed.

When the user presses the Print button and image processing begins, the TFT is turned off. Once the image has been printed the TFT is turned on again.

Structural Overview

The Display Controller is used in those Artcam models that incorporate a flat panel display. An example display is a TFT LCD of resolution 240 x 160 pixels. The Display Controller has the following structure:

The Display Controller State Machine contains registers that control the timing of the Sync Generation, where the display image is to be taken from (in DRAM via the Cache Interface), and whether the TFT should be active or not (via TFT Enable) at the moment. The CPU can write to those registers via the low speed bus.

Displaying a 240 x 160 pixel image on an RGB TFT requires 3 components per pixel. The image taken from DRAM is displayed via 3 DACs, one for each of the R, G, and B output signals.

At an image refresh rate of 30 frames per second (60 fields per second) the Display Controller requires data transfer rates of:

240 x 160 x 3 x 30 = 3.5MB per second

This data rate is low compared to the rest of the system. However it is high enough to cause VLIW programs to slow down during the intensive image processing. The general princliples of TFT operation should reflect this.

CPU Core (CPU)

The CPU core 72 can be any processor core with sufficient processing power to perform the required core calculations and control functions fast enough to met consumer expectations. Examples of suitable cores are:

MIPS R4000 core from LSI Logic

StrongARM core

The Artcam is deliberately designed so that the core processor 72 can be changed at any stage while maintaining complete compatibility. To use a different core, the Vark interpreter and camera control programs must be re-compiled for the new processor instruction set. This is a straightforward task if the Vark interpreter is written in a high level language (preferably C++) with no assembler.

The Vark language preferably makes no assumptions about the CPU, and is completely portable. Therefore any Artcards will work with any CPU cores which meet the performance specifications. As a result of this device independence, future Artcam models can take advantage of new processor cores as they are developed. Also, different ACP chip designs may be fabricated by different manufacturers, without the need to license or port the CPU core.

Program Cache 75

A small cache 75 is required for good performance. This requirement is mostly due to the use of a Rambus DRAM, which can provide high-speed data in bursts, but is inefficient for single byte accesses. 16 dedicated cache lines of 32 bytes each will achieve most of the performance gain over no cache, and limits the cache size to 512 bytes.

Data Cache 76

As with the program cache 75, a small cache 76 is

required for good performance. This requirement is again mostly due to the use of a Rambus DRAM, which can provide high-speed data in bursts, but is inefficient for single byte accesses. 16 dedicated cache lines of 32 bytes each will achieve most of the performance gain over no cache, and limits the cache size to 512 bytes.

Image Sensor Interface (ISI) 83

The Image Sensor Interface (ISI) 83 takes data from the CCD and makes it available for storage in DRAM. The CCD can be is a 3:2 aspect ratio image sensor, typically 750 x 500, yielding 375K (8 bits per pixel). Fig. 24 illustrates the configuration of a single pixel.

As illustrated in simplified form in Fig.25, the ISI 83 includes a state machine that sends control information to the CCD 2 (Fig.2), including frame sync pulses and pixel clock pulses in order to read the image. Pixels are read from the CCD via a sub-ranging semi-flash DAC, and placed into the VLIW Input FIFO. The VLIW is then able to process and/or store the pixels, which are then available for processing and/or storage.

The ISI 83 is used in conjunction with a VLIW microcode program that stores the CCD image in DRAM. Processing occurs in 2 steps:

1. A small VLIW program reads the pixels from the FIFO 192 and writes them to the DRAM via a Sequential Write Iterator.
2. The CCD image in DRAM is rotated 90, 180 or 270 degrees according to the orientation of the camera when the photo was taken.

If the rotation is 0 degrees, then step 1 merely writes the CCD image out to the final CCD image location and step 2 is not performed. If the rotation is non-0 degrees, the image is written out to a temporary area (for example into the print image memory area), and then rotated during step 2 into the final CCD image location. Step 1 is very simple microcode, taking data from the VLIW Input FIFO 192 and

writing it to a Sequential Write Iterator. Step 2's rotation is accomplished by using the accelerated Vark Affine Transform function. The processing is performed in 2 steps in order to reduce design complexity and to re-use the Vark affine transform rotate logic already required for images. This is acceptable since both steps are completed in less than 0.03 seconds, a time imperceptible to the operator of the Artcam. Even so, the read process is CCD speed bound, taking 0.02 seconds to read the full frame. The time taken to rotate the image can be 2 cycles per output pixel, which is 750,000 cycles, or 0.008 seconds. The total time for both stages is therefore 0.028 seconds.

The orientation will be important for converting between the CCD image and the internal format image, since the relative positioning of R, G, and B pixels changes with orientation. The processed image may also have to be rotated during the Print process in order to be in the correct orientation for printing.

On the optional 3D model of the Artcam there are 2 CCDs, with their inputs multiplexed to a single ISI (different microcode, but same ACP). If the CCD has a frame store both frames can be taken simultaneously, and then transferred to memory one at a time. If the CCD has a line store, the frames can be transferred one line at a time in a multiplexed fashion.

Display Controller 88

The display controller 88 is used in those Artcam models that incorporate a flat panel display. An example display is a TFT LCD of resolution 240 x 160 pixels. This type of display would require a low data-rate.

When the "Take" button is half depressed, the TFT would display the current image from the image sensor. Once taken, the Taken Image would be displayed in its processed form.

Artcard Interface (AI) 87

The Artcard Interface (AI) 87 is responsible for taking an Artcard image from the Artcard Reader 34 , and decoding

it into the original data (usually a Vark script). Specifically, the AI 87 accepts signals from the Artcard scanner linear CCD 34 , detects the bit pattern printed on the card, and converts the bit pattern into the original data, correcting read errors.

With no Artcard 9 inserted, the image printed from an Artcam 30 is simply the sensed Photo Image cleaned up by any standard image processing routines. The Artcard 9 is the means by which users are able to modify a photo before printing it out. By the simple task of inserting a specific Artcard 9 into an Artcam 30, a user is able to define complex image processing to be performed on the Photo Image. With no Artcard 30 inserted the Photo Image is processed in a standard way to create the Print Image.

When a single Artcard 9 is inserted into the Artcam, that Artcard's effect is applied to the Photo Image to generate the Print Image.

When the Artcard 9 is removed (ejected), the printed image reverts to the Photo Image processed in a standard way.

When the user presses the button to eject an Artcard, an event is placed in the event queue maintained by the operating system running on the ACP72. When the event is processed (for example after the current Print has occurred), the following things occur:

If the current Artcard is valid, then the Print Image is marked as invalid and a 'Process Standard' event is placed in the event queue. When the event is eventually processed it will perform the standard image processing operations on the Photo Image to produce the Print Image.

The motor is started to eject the Artcard and a time-specific 'Stop-Motor' Event is added to the event queue.

Inserting an Artcard

When a user inserts an Artcard 9, the Artcard Sensor 49 detects it notifying the ACP72. This results in the software inserting an 'Artcard Inserted' event into the event queue. When the event is processed several things

occur:

The current Artcard is marked as invalid (as opposed to 'none').

The Print Image is marked as invalid.

The Artcard motor 37 is started up to load the Artcard

The Artcard Interface 87 is instructed to read the Artcard

The Artcard Interface 87 accepts signals from the Artcard scanner linear CCD 34, detects the bit pattern printed on the card, and corrects errors in the detected bit pattern, producing a valid Artcard data block in DRAM.

Reading Data from the Artcard CCD - General Considerations

As illustrated in Fig. 26, the Artcard 9 must be sampled at least at double the printed resolution to satisfy Nyquist's Theorem. In practice it is better to sample at a higher rate than this. Preferably, the pixels sampled at 3 times the resolution of a printed dot in each dimension, requiring 9 pixels to define a single dot eg 230. Thus if the resolution of the artcard 9 is 1600 dpi, and the resolution of the sensor 34 is 4800 dpi, then using a 50mm CCD image sensor results in 9450 pixels per column. Therefore if we require 2MB of dot data (at 9 pixels per dot) then this requires 2MB*8*9/9450 = 15,978 columns = approximately 16,000 columns. Of course if a dot is not exactly aligned with the sampling CCD the worst and most likely case is that a dot will be sensed over a 16 pixel area (4x4) 231.

An Artcard 9 may be slightly warped due to heat damage, slightly rotated (up to, say 1 degree) due to differences in insertion into an Artcard reader, and can have slight differences in true data rate due to fluctuations in the speed of the reader motor 37. These changes will cause columns of data from the card not to be read as corresponding columns of pixel data. As illustrated in Fig. 28, a 1 degree rotation in the Artcard 9 can cause the pixels from a column on the card to be read as pixels across

166 columns:

Finally, the Artcard 9 should be read in a reasonable amount of time with respect to the human operator. The data on the Artcard covers most of the Artcard surface, so we can limit our timing concerns to the Artcard data itself. A reading time of 1.5 seconds is adequate for Artcard reading.

The Artcard should be loaded in 1.5 seconds. Therefore all 16,000 columns of pixel data must be read from the CCD 34 in 1.5 second, i.e. 10,667 columns per second. Therefore the time available to read one column is 1/10667 seconds, or 93,747ns. Pixel data can be written to the DRAM 1 column at a time, completely independently from any processes that are reading the pixel data.

The time to write one column of data (9450/2 bytes since the reading can be 4 bits per pixel giving 2 x 4 bit pixels per byte) to DRAM is reduced by using 8 cache lines. If 4 lines were written out at one time, the 4 banks can be written to independently and thus overlap latency reduced. Thus the 4725 bytes can be written in 11,840ns (4725/128 * 320ns). Thus the time taken to write a given column's data to DRAM uses just under 13% of the available bandwidth.

Decoding an Artcard

A simple look at the data sizes shows the impossibility of fitting the process into the 8MB of memory 33 if the entire Artcard pixel data (140 MB if each bit is read as a 3x3 array) as read by the linear CCD 34 is kept. For this reason, the reading of the linear CCD, decoding of the bitmap, and the un-bitmap process should take place in real-time (while the Artcard 9 is travelling past the linear CCD 34), and these processes must effectively work without having entire data stores available.

When an Artcard 9 is inserted, the old stored Print Image and any expanded Photo Image becomes invalid. The new Artcard 9 can contain directions for creating a new image based on the currently captured Photo Image. The old Print Image is invalid, and the area holding expanded Photo Image

data and image pyramid is invalid, leaving more than 5MB that can be used as scratch memory during the read process. Strictly speaking, the 1MB area where the Artcard raw data is to be written can also be used as scratch data during the Artcard read process as long as by the time the final Reed-Solomon decode is to occur, that 1MB area is free again. The reading process described here does not make use of the extra 1MB area (except as a final destination for the data).

It should also be noted that the unscrambling process requires two sets of 2MB areas of memory since unscrambling cannot occur in place. Fortunately the 5MB scratch area contains enough space for this process.

Turning now to Fig. 27, there is shown a flowchart 220 of the steps necessary to decode the Artcard data. These steps include reading in the artcard 221, decoding the read data to produce corresponding encoded XORed scrambled bitmap data 223. Next a checkerboard XOR is applied to the data to produces encoded scrambled data 224. This data is then unscrambled 227 to produce data 225 before this data is subjected to Reed-Solomon decoding to produce the original raw data 226. Alternatively, unscrambling and XOR process can take place together, not requiring a separate pass of the data. Each of the above steps is discussed in further detail hereinafter. The Artcard Interface, therefore, has 4 phases, the first 2 of which are time-critical, and must take place while pixel data is being read from the CCD:

Phase 1. Detect data area on Artcard

Phase 2. Detect bit pattern from Artcard based on CCD pixels, and write as bytes.

Phase 3. Descramble and XOR the byte-pattern

Phase 4. Decode data (Reed-Solomon decode)

Fig. 29 illustrates a timeline 240 of the pixel reading process and the four phases which are as follows:

Phase 1. As the Artcard 9 moves past the CCD 34 the AI must detect the start of the data area by robustly detecting special targets on the Artcard to the left of the data area.

If these cannot be detected, the card is marked as invalid. The detection must occur in real-time, while the Artcard 9 is moving past the CCD 34.

Phase 2. Once the data area has been determined, the main read process begins, placing pixel data from the CCD into an 'Artcard data window', detecting bits from this window, assembling the detected bits into bytes, and constructing a byte-image in DRAM. This must all be done while the Artcard is moving past the CCD.

Phase 3. Once all the pixels have been read from the Artcard data area, the Artcard motor 37 can be stopped, and the byte image descrambled and XORed. Although not requiring real-time performance, the process should be fast enough not to annoy the human operator. The process must take 2 MB of scrambled bit-image and write the unscrambled/XORed bit-image to a separate 2MB image.

Phase 4. The final phase in the Artcard read process is the Reed-Solomon decoding process, where the 2MB bit-image is decoded into a 1MB valid Artcard data area. Again, while not requiring real-time performance it is still necessary to decode quickly with regard to the human operator. If the decode process is valid, the card is marked as valid. If the decode failed, any duplicates of data in the bit-image are attempted to be decoded, a process that is repeated until success or until there are no more duplicate images of the data in the bit image.

The 4 phase process described requires 4.5 MB of DRAM. 2MB is reserved for Phase 2 output, and 0.5MB is reserved for scratch data during phases 1 and 2. The remaining 2MB of space can hold over 440 columns at 4725 byes per column. In practice, the pixel data being read is a few columns ahead of the phase 1 algorithm, and in the worst case, about 180 columns behind phase 2, comfortably inside the 440 column limit.

A description of the actual operation of each phase will now be provided in greater detail.

Phase 1 - Detect data area on Artcard

This phase is concerned with robustly detecting the left-hand side of the data area on the Artcard 9. Accurate detection of the data area is achieved by accurate detection of special targets printed on the left side of the card. These targets are especially designed to be easy to detect even if rotated up to 1 degree.

Turning to Fig.30, there is shown an enlargement of the left hand side of an Artcard 9. The side of the card is divided into 16 bands, eg with a target 241 located at the center of each band. The bands are logical in that there is no line 242 drawn to separate bands. Turning to Fig.31, there is shown a single target 241. The target 241, is a printed black square containing a single white dot. The idea is to detect firstly as many targets 241 as possible, and then to join at least 8 of the detected white-dot locations into a single logical straight line. If this can be done 243 is set, the data area is a fixed distance from this logical line. If it cannot be done, then the card is rejected as invalid.

Returning to Fig. 30, the height of the card 9 is 3150 dots. A target (Target0) 241 is placed a fixed distance of 24 dots away from the top left corner 244 of the data area so that it falls well within the first of 16 equal sized regions 239 of 192 dots (576 pixels) with no target in the final pixel region of the card 9 . The target 241 must be big enough to be easy to detect, yet be small enough not to go outside the height of the region if the card is rotated 1 degree. A suitable size for the target is a 31 x 31 dot (93 x 93 pixels) black square 241 with the white dot 242.

At the worst rotation of 1 degree, we get a 1 column shift every 57 pixels. Therefore in a 590 pixel sized band, we cannot place any part of our symbol in the top or bottom 12 pixels or so of the band or they could be detected in the wrong band at CCD read time if the card is worst case rotated.

Therefore, if the black part of the rectangle is 57 pixels high (19 dots) we can be sure that at least 9.5 black pixels will be read in the same column by the CCD (worst case is half the pixels are in one column and half in the next). To be sure of reading at least 10 black dots in the same column, we must have a height of 20 dots. To give room for erroneous detection on the edge of the start of the black dots, we increase the number of dots to 31, giving us 15 on either side of the white dot at the target's local coordinate (15, 15). 31 dots is 91 pixels, which at most suffers a 3 pixel shift in column, easily within the 576 pixel band.

Thus each target is a block of 31 x 31 dots (93 x 93 pixels) each with the composition:

15 columns of 31 black dots each (45 pixel width columns of 93 pixels).

1 column of 15 black dots (45 pixels) followed by 1 white dot (3 pixels) and then a further 15 black dots (45 pixels)

15 columns of 31 black dots each (45 pixel width columns of 93 pixels)

Detect targets

Targets are detected by reading columns of pixels, one column at a time rather than by detecting dots. It is necessary to look within a given band for a number of columns consisting of large numbers of contiguous black pixels to build up the left side of a target. Next, it is expected to see a white region in the center of further black columns, and finally the black columns to the left of the target center.

Eight cache lines are required for good cache performance on the reading of the pixels. Each logical read fills 4 cache lines via 4 sub-reads while the other 4 cache-lines are being used. This effectively uses up 13% of the available RDRAM bandwidth.

As illustrated in Fig.33, the detection mechanism FIFO

for detecting the targets uses a filter 245, run-length encoder 246, and a FIFO 247 that requires special wiring of the top 3 elements (S1, S2, and S3) for random access.

The columns of input pixels are processed one at a time until either all the targets are found, or until a specified number of columns have been processed. To process a column the pixels are read from DRAM, passed through a filter 245 to detect a 0 or 1, and then run length encoded 246. The bit value and the number of contiguous bits of the same value are placed inFIFO FIFOthe FIFO 247. Each entry of the FIFO 249 is in 8 bits. 7 bits 50 to hold the run-length, and 1 bit 249 to hold the value of the bit detected.

The run-length encoder 246 only encodes contiguous pixels within a 576 pixel (192 dot) region.

The top 3 elements in the FIFO 247 can be accessed 252 in any random order. The run lengths (in pixels) of these entries are filtered into 3 values: *short*, *medium*, and *long* in accordance with the following table:

| | | |
|---|---|---|
| Short | Used to detect white dot. | RunLength < 16 |
| Medium | Used to detect runs of black above or below the white dot in the center of the target. | 16<= RunLength < 48 |
| Long | Used to detect run lengths of black to the left and right of the center dot in the target. | RunLength >= 48 |

Looking at the top three entries in the FIFO 247 there are 3 specific cases of interest:

| | | |
|---|---|---|
| Case 1 | S1 = white long<br>S2 = black long<br>S3 = white | We have detected a black column of the target to the left of or to the right of |

| | | |
|---|---|---|
| | medium/long | the white center dot. |
| Case 2 | S1 = white long<br>S2 = black medium<br>S3 = white short<br>Previous 8 columns were Case 1 | If we've been processing a series of columns of Case 1s, then we have probably detected the white dot in this column. We know that the next entry will be black (or it would have been included in the white S3 entry), but the number of black pixels is in question. Need to verify by checking after the next FIFO advance (see Case 3). |
| Case 3 | Prev = Case 2<br>S3 = black med | We have detected part of the white dot. We expect around 3 of these, and then some more columns of Case 1. |

Preferably, the following information per region band is kept:

| | |
|---|---|
| TargetDetected | 1 bit |
| BlackDetectCount | 4 bits |
| WhiteDetectCount | 3 bits |
| PrevColumnStartPixel | 15 bits |
| TargetColumnordinate | 16 bits (15:1) |
| TargetRowordinate | 16 bits (15:1) |
| TOTAL | 7 bytes (rounded to 8 bytes for easy addressing) |

Given a total of 7 bytes, it makes address generation easier if the total is assumed to be 8 bytes. Thus 16 entries requires 16 * 8 = 128 bytes, which fits in 4 cache lines. The address range would be inside the scratch 0.5MB DRAM area since other phases make use of the remaining 4MB data area.

When beginning to process a given pixel column, the register value S2StartPixel 254 is reset to 0. As entries in the FIFO advance from S2 to S1, they are also added 255 to the existing S2StartPixel value, giving the exact pixel position of the run currently defined in S2. Looking at each of the 3 cases of interest in the FIFO, S2StartPixel can be used to determine the start of the black area of a target (Cases 1 and 2), and also the start of the white dot in the center of the target (Case 3). An algorithm for processing columns can be as follows:

| | |
|---|---|
| 1 | TargetDetected[0-15] := 0<br>BlackDetectCount[0-15] := 0<br>WhiteDetectCount[0-15] := 0<br>TargetRow[0-15] := 0<br>TargetColumn[0-15] := 0<br>PrevColStartPixel[0-15] := 0<br>CurrentColumn := 0 |
| 2 | Do ProcessColumn |
| 3 | CurrentColumn++ |
| 4 | If (CurrentColumn <= LastValidColumn)<br>Goto 2 |

The steps involved in the processing a column (Process Column) are as follows:

| | |
|---|---|
| 1 | S2StartPixel := 0<br>FIFO := 0<br>BlackDetectCount := 0 |

| | |
|---|---|
| | WhiteDetectCount := 0<br>ThisColumnDetected := FALSE<br>PrevCaseWasCase2 := FALSE |
| 2 | If (! TargetDetected[Target]) & (! ColumnDetected[Target])<br>ProcessCases<br>EndIf |
| 3 | PrevCaseWasCase2 := Case=2 |
| 4 | Advance FIFO |

The processing for each of the 3 (Process Cases) cases is as follows:

Case 1:

| | |
|---|---|
| BlackDetectCount[target] < 8<br>OR<br>WhiteDetectCount[Target] = 0 | Δ := ABS(S2StartPixel – PrevColStartPixel[Target])<br>If (0<=Δ< 2)<br>BlackDetectCount[Target]++ (max value =8)<br>Else<br>BlackDetectCount[Target] := 1<br>WhiteDetectCount[Target] := 0<br>EndIf<br>PrevColStartPixel[Target] := S2StartPixel<br>ColumnDetected[Target] := TRUE<br>BitDetected = 1 |
| BlackDetectCount[target] >= 8<br>WhiteDetectCount[Target] != 0 | PrevColStartPixel[Target] := S2StartPixel<br>ColumnDetected[Target] := TRUE<br>BitDetected = 1<br>TargetDetected[Target] := TRUE<br>TargetColumn[Target] := CurrentColumn – 8 – (WhiteDetectCount[Target]/2) |

Case 2:

No special processing is recorded except for setting the 'PrevCaseWasCase2' flag for identifying Case 3 (see Step 3 of processing a column described above)

Case 3:

| | |
|---|---|
| PrevCaseWasCase2 = TRUE<br>BlackDetectCount[Target] >= 8<br>WhiteDetectCount=1 | If (WhiteDetectCount[Target] < 2)<br>TargetRow[Target] = S2StartPixel + ($S2_{RunLength}$/2)<br>EndIf<br>Δ := ABS(S2StartPixel – PrevColStartPixel[Target])<br>If (0<=Δ< 2)<br>WhiteDetectCount[Target]++<br>Else<br>WhiteDetectCount[Target] := 1<br>EndIf<br>PrevColStartPixel[Target] := S2StartPixel<br>ThisColumnDetected := TRUE<br>BitDetected = 0 |

At the end of processing a given column, a comparison is made of the current column to the maximum number of columns for target detection. If the number of columns allowed has been exceeded, then it is necessary to check how many targets have been found. If fewer than 8 have been found, the card is considered invalid.

Process targets

After the targets have been detected, they should be processed. All the targets may be available or merely some of them. Some targets may also have been erroneously detected.

This phase of processing is to determine a mathematical line that passes through the center of as many targets as possible. The more targets that the line passes through, the more confident the target position has been found. The limit is set to be 8 targets. If a line passes through at least 8 targets, then it is taken to be the right one.

It is alright to take a brute-force but straightforward approach since there is the time to do so (see below), and lowering complexity makes testing easier. It is necessary to

determine the line between targets 0 and 1 (if both targets are considered valid) and then determine how many targets fall on this line. Then we determine the line between targets 0 and 2, and repeat the process. Eventually we do the same for the line between targets 1 and 2, 1 and 3 etc and finally for the line between targets 14 and 15. Assuming all the targets have been found, we need to perform 15+14+13+ ...= 90 sets of calculations (with each set of calculations requiring 16 tests = 1440 actual calculations), and choose the line which has the maximum number of targets found along the line. The algorithm for target location can be as follows:

```
TargetA := 0
     MaxFound := 0
     BestLine := 0
     While (TargetA < 15)
          If (TargetA is Valid)
          TargetB:= TargetA + 1
          While (TargetB<= 15)
          If (TargetB is valid)
          CurrentLine := line between TargetA and TargetB
          TargetC := 0;
          While (TargetC <= 15)
               If (TargetC valid AND TargetC on line AB)
                    TargetsHit++
               EndIf
               If (TargetsHit > MaxFound)
                    MaxFound := TargetsHit
                    BestLine := CurrentLine
               EndIf
               TargetC++
          EndWhile
               EndIf
     TargetB ++
          EndWhile
     EndIf
```

```
        TargetA++
    EndWhile

    If (MaxFound < 8)
        Card is Invalid
    Else
        Store expected centroids for rows based on BestLine
    EndIf
```

As illustrated in Fig. 33, in the algorithm above, to determine a CurrentLine 260 from Target A 261 and target B, it is necessary to calculate Δrow 264 & Δcolumn 265 between targets 261, 262, and the location of Target A. It is then possible to move from Target 0 to Target 1 etc by adding r and Δcolumn . The found (if actually found) location of target N can be compared to the calculated expected position of Target N on the line, and if it falls within the tolerance, then Target N is determined to be on the line.

To calculate Δrow & D column:

$$\Delta row = (row_{TargetA} - row_{TargetB}) / (B-A)$$

$$\Delta column = (column_{TargetA} - column_{TargetB}) / (B-A)$$

Then we calculate the position of Target0:

row = rowTargetA - (A * Δrow)

column = columnTargetA - (A * Δcolumn )

And compare (row, column) against the actual $row_{Target0}$ and $column_{Target0}$. To move from one expected target to the next (e.g. from Target0 to Target1), we simply add Δrow and Δcolumn to row and column respectively. To check if each target is on the line, we must calculate the expected position of Target0, and then perform one add and one comparison for each target ordinate.

At the end of comparing all 16 targets against a maximum of 90 lines, the result is the best line through the valid targets. If that line passes through at least 8 targets (i.e. MaxFound >= 8), it can be said that enough

targets have been found to form a line, and thus the card can be processed. If the best line passes through fewer than 8, then the card is considered invalid.

The resulting algorithm takes 180 divides to calculate Δrow and Δcolumn , 180 multiply/adds to calculate target0 position, and then 2880 adds/comparisons. The time we have to perform this processing is the time taken to read 36 columns of pixel data = 3,374,892ns. Not even accounting for the fact that an add takes less time than a divide, it is necessary to perform 3240 mathematical operations in 3,374,892ns. That gives approximately 1040ns per operation, or 104 cycles. The CPU can therefore safely perform the entire processing of targets, reducing complexity of design.

Update centroids based on data edge border and clockmarks

Step 0: Locate the data area

From Target 0 (241 of Fig.30) it is a predetermined fixed distance in rows and columns to the top left border 244 of the data area, and then a further 1 dot column to the vertical clock marks 273. So we use TargetA, Δrow and Δcolumn found in the previous stage (Δrow and Δcolumn refer to distances between targets) to calculate the centroid or expected location for Target0 as described previously.

Since the fixed pixel offset from Target0 to the data area is related to the distance between targets (192 dots between targets, and 24 dots between Target0 and the data area 243), simply add Δrow/8 to Target0's centroid column coordinate (aspect ratio of dots is 1:1). Thus the top co-ordinate can be defined as:

$$column_{DotColumnTop} = column_{Target0} + (\Delta row/8)$$

$$row_{DotColumnTop} = row_{Target}0 + (\Delta column\ /8)$$

Next Δrow and Δcolumn are updated to give the number of pixels between dots in a single column (instead of between targets) by dividing them by the number of dots

between targets:

$\Delta row = \Delta row/192$

$\Delta column = \Delta column/192$

We also set the currentColumn register (see Phase 2) to be -1 so that after step 2, when phase 2 begins, the currentColumn register will increment from -1 to 0.

Step 1: Write out the initial centroid deltas (D) and bit history

This simply involves writing setup information required for Phase 2.

This can be achieved by writing 0s to all the $\Delta row$ and $\Delta column$ entries for each row, and a bit history. The bit history is actually an expected bit history since it is known that to the left of the clock mark column 276 is a border column 277, and before that, a white area. The bit history therefore is 011, 010, 011, 010 etc.

Step 2: Update the centroids based on actual pixels read.

The bit history is set up in Step 1 according to the expected clock marks and data border. The actual centroids for each dot row can now be more accurately set (they were initially 0) by comparing the expected data against the actual pixel values. The centroid updating mechanism is achieved by simply performing step 3 of Phase 2.

Phase 2 - Detect bit pattern from Artcard based on pixels read, and write as bytes.

Since a dot from the Artcard 9 requires a minimum of 9 sensed pixels over 3 columns to be represented, there is little point in performing dot detection calculations every sensed pixel column. It is better to average the time required for processing over the average dot occurrence, and thus make the most of the available processing time. This allows processing of a column of dots from an Artcard 9 in the time it takes to read 3 columns of data from the Artcard. Although the most likely case is that it takes 4 columns to represent a dot, the $4^{th}$ column will be the last

column of one dot and the first column of a next dot. Processing should therefore be limited to only 3 columns.

As the pixels from the CCD are written to the DRAM in 13% of the time available, 83% of the time is available for processing of 1 column of dots i.e. 83% of (93,747*3) = 83% of 281,241ns = 233,430ns.

In the available time, it is necessary to detect 3150 dots, and write their bit values into the raw data area of memory. The processing therefore requires the following steps:

For each column of dots on the Artcard:

Step 0: Advance to the next dot column

Step 1: Detect the top and bottom of an Artcard dot column (check clock marks)

Step 2: Process the dot column, detecting bits and storing them appropriately

Step 3: Update the centroids

Since we are processing the Artcard's logical dot columns, and these may shift over 165 pixels, the worst case is that we cannot process the first column until at least 165 columns have been read into DRAM. Phase 2 would therefore finish the same amount of time after the read process had terminated. The worst case time is: 165 * 93,747ns = 15,468,255ns or 0.015 seconds.

Step 0: Advance to the next dot column

In order to advance to the next column of dots we add $\Delta row$ and $\Delta column$ to the dotColumnTop to give us the centroid of the dot at the top of the column. The first time we do this, we are currently at the clock marks column 276 to the left of the bit image, and so we advance to the first column of data. Since $\Delta row$ and $\Delta column$ refer to distance between dots within a column, to move between dot columns it is necessary to add $\Delta row$ to $column_{dotColumnTop}$ and $\Delta column$ to:

$row_{dotColumnTop}$.

To keep track of what column number is being processed,

the column number is recorded in a register called CurrentColumn. Every time the sensor advances to the next dot column it is necessary to increment the CurrentColumn register. The first time it is incremented, it is incremented from -1 to 0 (see Step 0 Phase 1). The CurrentColumn register determines when to terminate the read process (when reaching maxColumns), and also is used to advance the DataOut Pointer to the next column of byte information once all 8 bits have been written to the byte (once every 8 dot columns). The lower 3 bits determin what bit we're up to within the current byte. It will be the same bit being written for the whole column.

Step 1: Detect the top and bottom of an Artcard dot column.

In order to process a dot column from an Artcard, it is necessary to detect the top and bottom of a column. The column should form a straight line between the top and bottom of the column (except for local warping etc). Initially dotColumnTop points to the clock mark column 276, we simply toggle the expected value, write it out into the bit history, and move on to step 2, whose first task will be to add the $\Delta$row and $\Delta$column values to dotColumnTop to arrive at the first data dot of the column.

Step 2: Process an Artcard's dot column

Given the centroids of the top and bottom of a column in pixel coordinates the column should form a straight line between them, with possible minor variances due to warping etc.

Assuming the processing is to start at the top of a column (at the top centroid coordinate) and move down to the bottom of the column, subsequent expected dot centroids are given as:

$$row_{next} = row + \Delta row$$

$$column_{next} = column + \Delta column$$

This gives us the address of the expected centroid for

the next dot of the column. However to account for *local* warping and error we add another Δrow and Δcolumn based on the last time we found the dot in a given row. In this way we can account for small drifts that accumulate into a maximum drift of some percentage from the straight line joining the top of the column to the bottom.

We therefore keep 2 values for each row, but store them in separate tables since the row history is used in step 3 of this phase.

* Δrow and Δcolumn (2 @ 4 bits each = 1 byte)

* row history (3 bits per row, 2 rows are stored per byte)

For each row we need to read a Δrow and Δcolumn to determine the change to the centroid. The read process takes 5% of the bandwidth and 2 cache lines:

76*(3150/32) + 2*3150 = 13,824ns = 5% of bandwidth

Once the centroid has been determined, the pixels around the centroid need to be examined to detect the status of the dot and hence the value of the bit. In the worst case a dot covers a 4x4 pixel area. However, thanks to the fact that we are sampling at 3 times the resolution of the dot, the number of pixels required to detect the status of the dot and hence the bit value is much less than this. We only require access to 3 columns of pixel columns at any one time.

In the worst case of pixel drift due to a 1% rotation, centroids will shift 1 column every 57 pixel rows, but since a dot is 3 pixels in diameter, a given column will be valid for 171 pixel rows (3*57). As a byte contains 2 pixels, the number of bytes valid in each buffered read (4 cache lines) will be a worst case of 86 (out of 128 read).

Once the bit has been detected it must be written out to DRAM. We store the bits from 8 columns as a set of contiguous bytes to minimise DRAM delay. Since all the bits from a given dot column will correspond to the next bit

position in a data byte, we can read the old value for the byte, shift and OR in the new bit, and write the byte back.

The read / shift&OR / write process requires 2 cache lines.

We need to read and write the bit history for the given row as we update it. We only require 3 bits of history per row, allowing the storage of 2 rows of history in a single byte. The read / shift&OR / write process requires 2 cache lines.

The total bandwidth required for the bit detection and storage is summarized in the following table:

| | |
|---|---|
| Read centroid Δ | 5% |
| Read 3 columns of pixel data | 19% |
| Read/Write detected bits into byte buffer | 10% |
| Read/Write bit history | 5% |
| | |
| TOTAL | 39% |

Detecting a dot

The process of detecting the value of a dot (and hence the value of a bit) given a centroid is accomplished by examining 3 pixel values and getting the result from a lookup table. The process is fairly simple and is illustrated in Fig. 34. A dot 290 has a radius of 1.5 pixels. Therefore the pixel 291 that holds the centroid, regardless of the actual position of the centroid within that pixel, should be 100% of the dot's value. If the centroid is exactly in the center of the pixel 291, then the pixels above 292 & below 293 the centroid's pixel, as well as the pixels to the left 294 & right 295 of the centroid's pixel will contain a majority of the dot's value. The further a centroid is away from the exact center of the pixel 295, the more likely that more than the center pixel will have 100% coverage by the dot.

Although Fig. 34 only shows centroids differing to the left and below the center, the same relationship obviously holds for centroids above and to the right of center. center. In Case 1, the centroid is exactly in the center of the middle pixel 295. The center pixel 295 is completely covered by the dot, and the pixels above, below, left, and right are also well covered by the dot. In Case 2, the centroid is to the left of the center of the middle pixel 291. The center pixel is still completely covered by the dot, and the pixel 294 to the left of the center is now completely covered by the dot. The pixesl above 292 and below 293 are still well covered. In Case 3, the centroid is below the center of the middle pixel 291. The center pixel 291 is still completely covered by the dot 291, and the pixel below center is now completely covered by the dot. The pixels left 294 and right 295 of center are still well covered. In Case 4, the centroid is left and below the center of the middle pixel. The center pixel 291 is still completely covered by the dot, and both the pixel to the left of center 294 and the pixel below center 293 are completely covered by the dot.

The algorithm for updating the centroid uses the distance of the centroid from the center of the middle pixel 291 in order to select 3 representative pixels and thus decide the value of the dot:

Pixel 1: the pixel containing the centroid

Pixel 2: the pixel to the left of Pixel 1 if the centroid's X coordinate (column value) is < ½, otherwise the pixel to the right of Pixel 1.

Pixel 3: the pixel above pixel 1 if the centroid's Y coordinate (row value) is < ½, otherwise the pixel below Pixel 1.

As shown in Fig.35, the value of each pixel is output to a precalculated lookup table 301. The 3 pixels are fed into a 12-bit lookup table, which outputs a single bit indicating the value of the dot - on or off. The lookup

table 301 is constructed at chip definition time, and can be compiled into about 500 gates. The lookup table can be a simple threshold table, with the exception that the center pixel (Pixel 1) is weighted more heavily.

Step 3: Update the centroid Δs for each row in the column

The idea of the Δs processing is to use the previous bit history to generate a 'perfect' dot at the expected centroid location for each row in a current column. The actual pixels (from the CCD) are compared with the expected 'perfect' pixels. If the two match, then the actual centroid location must be exactly in the expected position, so the centroid Δs must be valid and not need updating. Otherwise a process of changing the centroid Δs needs to occur in order to best fit the expected centroid location to the actual data. The new centroid Δs will be used for processing the dot in the next column.

Updating the centroid Δs is done as a subsequent process from Step 2 for the following reasons:

to reduce complexity in design, so that it can be performed as Step 2 of Phase 1 there is enough bandwidth remaining to allow it to allow reuse of DRAM buffers, and to ensure that all the data required for centroid updating is available at the start of the process without special pipelining.

The centroid Δ are processed as Δcolumn Δrow respectively to reduce complexity.

Although a given dot is 3 pixels in diameter, it is likely to occur in a 4x4 pixel area. However the edge of one dot will as a result be in the same pixel as the edge of the next dot. For this reason, centroid updating requires more than simply the information about a given single dot.

Fig.36 shows a single dot 310 from the previous column with a given centroid 311. In this example, the dot 310 extend Δ over 4 pixel columns 312-315 and in fact, part of

the previous dot column's dot (coordinate = (Prevcolumn, Current Row) has entered the current column for the dot on the current row. If the dot in the current row and column was white, we would expect the rightmost pixel column 314 from the previous dot column to be a low value, since there is only the dot information from the previous column's dot (the current column's dot is white). From this we can see that the higher the pixel value is in this pixel column 315, the more the centroid should be to the right Of course, if the dot to the right was also black, we cannot adjust the centroid as we cannot get information sub-pixel. The same can be said for the dots to the left, above and below the dot at dot coordinates (PrevColumn, CurrentRow).

From this we can say that a maximum of 5 pixel columns and rows are required. It is possible to simplify the situation by taking the cases of row and column centroid Δs separately, treating them as the same problem, only rotated 90 degrees.

Taking the horizontal case first, it is necessary to change the column centroid Δs if the expected pixels don't match the detected pixels. From the bit history, the value of the bits found for the Current Row in the current dot column, the previous dot column, and the (previous-1)th dot column are known. The expected centroid location is also known. Using these two pieces of information, it is possible to generate a 20 bit expected bit pattern should the read be 'perfect'. The 20 bit bit-pattern represents the expecteΔ for each of the 5 pixels across the horizontal dimension. The first nybble would represent the rightmost pixel of the leftmost dot. The next 3 nybbles represent the 3 pixels across the center of the dot 310 from the previous column, and the last nybble would be the leftmost pixel 317 of the rightmost dot (from the current column).

If the expected centroid is in the center of the pixel, we would expect a 20 bit pattern based on the following

table:

| Bit history | Expected pixels |
|---|---|
| 000 | 00000 |
| 001 | 0000D |
| 010 | 0DFD0 |
| 011 | 0DFDD |
| 100 | D0000 |
| 101 | D000D |
| 110 | DDFD0 |
| 111 | DDFDD |

The pixels to the left and right of the center dot are either 0 or D depending on whether the bit was a 0 or 1 respectively. The center three pixels are either 000 or DFD depending on whether the bit was a 0 or 1 respectively. These values are based on the physical area taken by a dot for a given pixel. Depending on the distance of the centroid from the exact center of the pixel, we would expect data shifted slightly, which really only affects the pixels either side of the center pixel. Since there are 16 possibilities, it is possible to divide the distance from the center by 16 and use that amount to shift the expected pixels.

Once the 20 bit 5 pixel expected value has been determined it can be compared against the actual pixels read. This can proceed by subtracting the expected pixels from the actual pixels read on a pixel by pixel basis, and finally adding the differences together to obtain a distance from the expecteΔ.

Turning to Fig.37, there is illustrated one form of implementation of the above algorithm which includes a look up table 320 which receives the bit history 322 and central fractional component 323 and outputs 324 the corresponding 20 bit number which is subtracted 321 from the central pixel input 326 to produce a pixel difference 327.

This process is carried out for the expected centroid and once for a shift of the centroid left and right by 1 amount in Δcolumn . The centroid with the smallest difference from the actual pixels is considered to be the 'winner' and the Δcolumn updated accordingly (which hopefully is 'no change'). As a result, a Δcolumn cannot change by more than 1 each dot column.

The process is repeated for the vertical pixels, and Δrow is consequentially updated.

There is a large amount of scope here for parallelism. Depending on the rate of the clock chosen for the ACP unit 31 these units can be placed in series (and thus the testing of 3 different Δ could occur in consecutive clock cycles), or in parallel where all 3 can be tested simultaneously. If the clock rate is fast enough, there is less need for parallelism.

Bandwidth utilization

It is necessary to read the old Δ of the Δs, and to write them out again. This takes 10% of the bandwidth:

2 * (76(3150/32) + 2*3150) = 27,648ns = 10% of bandwidth

It is necessary to read the bit history for the given row as we update its Δs. Each byte contains 2 row's bit histories, thus taking 2.5% of the bandwidth:

76((3150/2)/32) + 2*(3150/2) = 4,085ns = 2.5% of bandwidth

In the worst case of pixel drift due to a 1% rotation, centroids will shift 1 column every 57 pixel rows, but since a dot is 3 pixels in diameter, a given pixel column will be valid for 171 pixel rows (3*57). As a byte contains 2 pixels, the number of bytes valid in cached reads will be a worst case of 86 (out of 128 read). The worst case timing for 5 columns is therefore 31% bandwidth.

5 *(((9450/(128 * 2)) * 320) * 128/86) = 88, 112ns = 31% of bandwidth.

The total bandwidth required for the updating the centroid Δ is summarized in the following table:

| | |
|---|---|
| Read/Write centroid Δ | 10% |
| Read bit history | 2.5% |
| Read 5 columns of pixel data | 31% |
| | |
| TOTAL | 43.5% |

Summary of Bandwidth for Phase 2

The total bandwidth required for the phase 2 is summarized in the following table:

| | |
|---|---|
| Step 0 | 0% |
| Step 1 | 0.5% |
| Step 2 | 39% |
| Step 3 | 43.5<br>.5 |
| TOTAL | 83% |

The reading of the pixel data from the CCD occurs at the same, and uses 13% of available bandwidth. This combines for a total of 96%.

Memory usage for Phase 2:

The 2MB bit-image DRAM area is read from and written to during Phase 2 processing. The 2MB pixel-data DRAM area is read.

The 0.5MB scratch DRAM area is used for storing row data, namely:

| | |
|---|---|
| Centroid array | 24bits (16:8) * 2 * 3150 = 18,900 byes |
| Bit History array | 3 bits * 3150 entries (2 per byte) = 1575 bytes |

Phase 3 -Unscramble and XOR the raw data

Returning to Fig.28, the next step in decoding is to

unscramble and XOR the raw data. The 2MB byte image, as taken from the Artcard, is in a scrambled XORed form. It must be unscrambled and re-XORed to retrieve the bit image necessary for the Reed Solomon decoder in phase 4.

Turning to Fig.38, the unscrambling process 330 takes a 2MB scrambled byte image 331 and writes an unscrambled 2MB image 332. The process cannot reasonably be performed in-place, so 2 sets of 2MB areas are utilised. The scrambled data 331 is in symbol block order arranged in a 16x16 array, with symbol-block 0 (334) having all the symbol 0's from all the code words in random order. Symbol block 1 has all the symbol 1's from all the code words in random order etc. Since there are only 255 symbols, the 256$^{th}$ symbol block is currently unused.

A linear feedback shift register is used to determine the relationship between the position within a symbol block eg. 334 and what code word eg. 355 it came from. *This works as long as the same seed is used when generating the original Artcard images.* The XOR of bytes from alternative source lines with 0xAA and 0x55 respectively is effectively free (in time) since the bottleneck of time is waiting for the DRAM to be ready to read/write to non-sequential addresses.

The timing of the unscrambling XOR process is effectively 2MB of random byte-reads, and 2MB of random byte-writes i.e. 2 * (2MB * 76ns + 2MB * 2ns) = 327,155,712ns or approximately 0.33 seconds. This timing assumes no caching.

Phase 4 - Reed Solomon decode

This phase is a loop, iterating through copies of the data in the bit image, passing them to the Reed-Solomon decode module until either a successful decode is made or until there are no more copies to attempt decode from.

The Reed-Solomon decoder used is a core such as LSI Logic's L64712.

The L64712 has a throughput of 50Mbits per second

(around 6.25MB per second), so the time is bound by the speed of the Reed-Solomon decoder rather than the 2MB read and 1 MB write memory access time (500MB/sec for sequential accesses). The time taken in the worst case is thus 2/6.25s = approximately 0.32 seconds.

Phase 5 Running the Vark script

The overall time taken to read the Artcard 9 and decode it is therefore approximately 2.15 seconds. The apparent delay to the user is actually only 0.65 seconds (the total of Phases 3 and 4), since the Artcard stops moving after 1.5 seconds.

Once the Artcard is loaded, the Artvark script must be interpreted, Rather than run the script immediately, the script is only run upon the pressing of the 'Print' button 13 (Fig.1). Time taken to run the script will vary depending on the complexity of the script, and must be taken into account for the perceived delay between pressing the print button and the actual print button and the actual printing.

Vark Accelerator 79

The Vark Accelerator (VA) 79 (Fig.3) is a digital processing system that accelerates computationally expensive Vark functions. The balance of functions performed in software by the CPU core 72, and in hardware by the Vark accelerator 79 which is implementation dependent. The goal of the VA 79 is to assist all Artcard styles to execute in a time that does not seem to slow to the user. As CPUs become faster and more powerful, the number of functions requiring hardware acceleration becomes less and less. The ACP has a microcoded ALU sub-system that allows general hardware speedup of the following time-critical functions.

1) Image access mechanisms for general software processing
2) Image convolver.
3) Data driven image warper
4) Image scaling
5) Image tessellation

6) Affine transform
7) Image compositor
8) Colour space transform
9) Histogram collector
10) Illumination of the Image
11) Brush stamper
12) Histogram collector
13) CCD image to internal image conversion
14) Construction of image pyramids (used by warper & for brushing)

The following table summarizes the time taken for each Vark operation if implemented in the ALU model. The method of implementing the function using the ALU model is described hereinafter.

| Operation | Speed of Operation | 1500 * 1000 image | |
|---|---|---|---|
| | | 1 channel | 3 channels |
| Image composite | 1 cycle per output pixel | 0.015 s | 0.045 s |
| Image convolve | k/3 cycles per output pixel (k = kernel size)<br>3x3 convolve<br>5x5 convolve<br>7x7 convolve | <br><br><br>0.045 s<br>0.125 s<br>0.245 s | <br><br><br>0.135 s<br>0.375 s<br>0.735 s |
| Image warp | 8 cycles per pixel | 0.120 s | 0.360 s |
| Histogram collect | 2 cycles per pixel | 0.030 s | 0.090 s |
| Image Tessellate | 1/3 cycle per pixel | 0.005 s | 0.015 s |
| Image sub-pixel Translate | 1 cycle per output pixel | - | - |
| Color lookup | ½ cycle per | 0.008 s | 0.023 |

| | | | |
|---|---|---|---|
| replace | pixel | | |
| Color space transform | 8 cycles per pixel | 0.120 s | 0.360 s |
| Convert CCD image to internal image (including color convert & scale) | 4 cycles per output pixel | 0.06 s | 0.18 s |
| Construct image pyramid | 1 cycle per input pixel | 0.015 s | 0.045 s |
| Scale | Maximum of:<br>2 cycles per input pixel<br>2 cycles per output pixel<br>2 cycles per output pixel (scaled in X only) | 0.015 s (minimum) | 0.045 s (minimum) |
| Affine transform | 2 cycles per output pixel | 0.03 s | 0.09 s |
| Brush rotate/translate and composite | ? | | |
| Tile Image | 4-8 cycles per output pixel | 0.015 s to 0.030 s | 0.060 s to 0.120 s to for 4 channels (Lab, texture) |
| Illuminate image<br>Ambient only<br>Directional | Cycles per pixel<br>½<br>1 | <br>0.008 s<br>0.015 s<br>0.09 s | <br>0.023 s<br>0.045 s<br>0.27 s |

| | | | |
|---|---|---|---|
| light | 6 | 0.09 s | 0.27 s |
| Directional | 6 | 0.137 s | 0.41 s |
| (bm) | 9 | 0.137 s | 0.41 s |
| Omni light | 9 | 0.18 s | 0.54 s |
| Omni (bm) | 12 | | |
| Spotlight | | | |
| Spotlight (bm) | | | |
| (bm) = bumpmap | | | |

For example, to convert a CCD image, collect histogram & perform lookup-colour replacement (for image enhancement) takes: 9+2+0.5 cycles per pixel, or 11.5 cycles. For a 1500 x 1000 image that is 172,500,000, or approximately 0.2 seconds per component, or 0.6 seconds for all 3 components. Add a simple warp, and the total comes to 0.6 + 0.36, almost 1 second.

Image Convolver

A convolve is a weighted average around a center pixel. The average may be a simple sum, a sum of absolute values, the absolute value of a sum, or sums truncated at 0.

The image convolver is a general-purpose convolver, allowing a variety of functions to be implemented by varying the values within a variable-sized coefficient kernel. The kernel sizes supported are 3x3, 5x5 and 7x7 only.

Turning now to Fig.39, there is illustrated 340 an example of the convolution process. The pixel component values fed into the convolver process 341 come from a Box Read Iterator 342. The Iterator 342 provides the image data row by row, and within each row, pixel by pixel. The output from the convolver 341 is sent to a Sequential Write Iterator 344, which stores the resultant image in a valid image format.

A Coefficient Kernel 346 is a lookup table in DRAM. The kernel is arranged with coefficients in the same order as

the Box Read Iterator 342. Each coefficient entry is 8 bits. A simple Sequential Read Iterator can be used to index into the kernel 346 and thus provide the coefficients. It simulates an image with ImageWidth equal to the kernel size, and a Loop option is set so that the kernel would continuously be provided.

One form of implementation of the convolve process is as illustrated in Fig. 40.The following constants are set by software:

| Constant | Value |
|---|---|
| $K_1$ | Kernel size (9, 25, or 49) |

The control logic is used to count down the number of multiply/adds per pixel. When the count (accumulated in $Latch_2$) reaches 0, the control signal generated is used to write out the current convolve value (from $Latch_1$) and to reset the count. In this way, one control logic block can be used for a number of parallel convolve streams.

With 3 parallel streams the requirements are summarized as follows:

| Requirements | *+ | + | K | LU | Iterators |
|---|---|---|---|---|---|
| General (convolve kernel) | 0 | 0 | 0 | 0 | 1 |
| General (per convolve stream) 1 | 1 | 0 | 1 | 0 | 2 |
| General (per convolve stream) 2 | 1 | 0 | 1 | 0 | 2 |
| General (per convolve stream) 3 | 1 | 0 | 1 | 0 | 2 |
| Control logic (one set required) | 0 | 1 | 2 | 0 | 0 |
| TOTAL | 3 | 1 | 5 | 0 | 7 |

Each cycle the multiply ALU can perform one multiply/add to incorporate the appropriate part of a pixel. The number of cycles taken to sum up all the values is therefore the number of entries in the kernel. Since this is compute bound, it is appropriate to divide the image into

multiple sections and process them in parallel.

On a 7x7 kernel, the time taken for each pixel is 49 cycles, or 490ns. Since each cache line holds 32 pixels, the time available for memory access is 12,740ns. ((32-7+1) x 490ns). The time taken to read 7 cache lines and write 1 is worse case 1,120ns (8*140ns, all accesses to same DRAM bank). Consequently it is possible to process up to 10 pixels in parallel given unlimited resources. Given a limited number of ALUs it is possible to do at best 4 in parallel. The time taken to therefore perform the convolution using a 7x7 kernel is 0.18375 seconds (1500*1000 * 490ns / 4 = 183,750,000ns).

On a 5x5 kernel, the time taken for each pixel is 25 cycles, or 250ns. Since each cache line holds 32 pixels, the time available for memory access is 7,000ns. ((32-5+1) x 250ns). The time taken to read 5 cache lines and write 1 is worse case 840ns (6 * 140ns, all accesses to same DRAM bank). Consequently it is possible to process up to 7 pixels in parallel given unlimited resources. Given a limited number of ALUs it is possible to do at best 4. The time taken to therefore perform the convolution using a 5x5 kernel is 0.09375 seconds (1500*1000 * 250ns / 4 = 93,750,000ns).

On a 3x3 kernel, the time taken for each pixel is 9 cycles, or 90ns. Since each cache line holds 32 pixels, the time available for memory access is 2,700ns. ((32-3+1) x 90ns). The time taken to read 3 cache lines and write 1 is worse case 560ns (4 * 140ns, all accesses to same DRAM bank). Consequently it is possible to process up to 4 pixels in parallel given unlimited resources. Given a limited number of ALUs and Read/Write Iterators it is possible to do at best 4. The time taken to therefore perform the convolution using a 3x3 kernel is 0.03375 seconds (1500*1000 * 90ns / 4 = 33,750,000ns).

Consequently each output pixel takes kernelsize/3 cycles to compute. The actual timings are summarized in the following

table:

| Kernel size | Time taken to calculate output pixel | Time to process 1 channel at 1500x1000 | Time to Process 3 channels at 1500x1000 |
|---|---|---|---|
| 3x3 (9) | 3 cycles | 0.045 seconds | 0.135 seconds |
| 5x5 (25) | 8 1/3 cycles | 0.125 seconds | 0.375 seconds |
| 7x7 (49) | 16 1/3 cycles | 0.245 seconds | 0.735 seconds |

Image Compositor

Compositing is to add a foreground image to a background image using a matte or a channel to govern the appropriate proportions of background and foreground in the final image. Two styles of compositing are preferably supported: regular compositing and associated compositing.

The rules for the two styles are:

Regular composite: new Value = Foreground + (Background - Foreground) a

Associated composite: new value = Foreground + (1-a) Background

The difference then, is that with associated compositing, the foreground has been pre-multiplied with the matte, while in regular compositing it has not. An example of the compositing process is as illustrated in Fig. 41.

*The a channel has values from 0 to 255 corresponding to the range 0 to 1.* Thus a regular composite is implemented as:

Regular Composite

A regular composite is implemented as:

Foreground + (Background - Foreground) * $\alpha$ / 255

The division by X/255 is approximated by 257X/65536. An implementation of the compositing process is shown in more detail in Fig. 42, where the following constant is set by software:

| Constant | Value |
|---|---|
| $K_1$ | 257 |

Since 4 Iterators are required, the composite process takes 1 cycle per pixel, with a utilization of only half of the ALUs. The composite process is only run on a single channel. To composite a 3-channel image with another, the compositor must be run 3 times, once for each channel.

The time taken to composite a full size single channel is 0.015s (1500 * 1000 * 1 * 10ns), or 0.045s to composite all 3 channels.

To approximate a divide by 255 it is possible to multiply by 257 and then divide by 65536. It can also be achieved by a single add (256 * x + x) and ignoring (except for rounding purposes) the final 16 bits of the result.

As shown in Fig. 41, the compositor process requires 3 Sequential Read Iterators 351-353 and 1 Sequential Write Iterator 355, and is implemented as microcode using 1 Adder ALU in conjunction with a multiplier ALU. Composite time is 1 cycle (10ns) per-pixel. Different microcode is required for associated and regular compositing, although the average time per pixel composite is the same.

The composite process is only run on a single channel. To composite one 3-channel image with another, the compositor must be run 3 times, once for each channel. As the a channel is the same for each composite, it must be read each time. However it should be noted that to transfer (read or write) 4 x 32 byte cache-lines in the best case takes 320ns. The pipeline gives an average of 1 cycle per pixel composite, taking 32 cycles or 320ns (at 100MHz) to composite the 32 pixels, so the a channel is effectively read for free. An entire channel can therefore be composited in:

1500/32 * 1000 * 320ns = 15,040,000ns = 0.015seconds.

The time taken to composite a full size 3 channel image is therefore 0.045 seconds.

Construct Image Pyramid

Several functions, such as warping, tiling and brushing, require the average value of a given area of pixels. Rather than calculate the value for each area given, these functions preferably make use of an image pyramid. As illustrated in Fig.42, an image pyramid 360 is effectively a multi-resolution pixelmap. The original image is a 1:1 representation. Sub-sampling by 2:1 in each dimension produces an image ¼ the original size. This process continues until the entire image is represented by a single pixel.

An image pyramid is constructed from an original image, and consumes 1/3 of the size taken up by the original image (1/4 + 1/16 + 1/64 + ...). For an original image of 1500 x 1000 the corresponding image pyramid is approximately ½ MB

The image pyramid is constructed via a 3x3 convolve performed on 1 in 4 input image pixels advancing the center of the convolve kernel by 2 pixels each dimension. A 3x3 convolve results in higher accuracy than simply averaging 4 pixels, and has the added advantage that coordinates on different pyramid levels differ only by shifting 1 bit per level.

The construction of an entire pyramid relies on a software loop that calls the pyramid level construction function once for each level of the pyramid.

The timing to produce 1 level of the pyramid is 9/4 * 1/4 of the resolution of the input image since we are generating an image 1/4 of the size of the original. Thus for a 1500 x 1000 image:

Timing to produce level 1 of pyramid = 9/4 * 750 * 500 = 843, 750 cycles

Timing to produce level 2 of pyramid = 9/4 * 375 * 250 = 210, 938 cycles

Timing to produce level 3 of pyramid = 9/4 * 188 * 125 = 52, 735 cycles

Etc.

The total time is 3/4 cycle per original image pixel

(image pyramid is 1/3 of original image size, and each pixel takes 9/4 cycles to be calculated, i.e. 1/3 * 9/4 = 3/4). In the case of a 1500 x 1000 image is 1,125,000 cycles (at 100MHz), or 0.011 seconds. This timing is for a single colour channel, 3 colour channels require 0.034 seconds processing time.

General Data Driven Image Warper

The ACP 31 is able to carry out image warping manipulations of the input image. The principles of image warping are well-known in theory. One thorough text book reference on the process of warping is "Digital Image Warping" by George Wolberg published in 1990 by the IEEE Computer Society Press, Los Alamitos, California. The warping process utilises a warp map which forms part of the data fed in via artcard 9. The warp map can be arbritarily dimensioned in accordance with requirements and provides information of a mapping of input pixels to output pixels. Unfortunately, the utilisation of arbritarily sized warp maps presents a number of problems which must be solved by the image warper.

Turning to Fig 43, a warp map 365, having dimensions AxB comprises array values of a certain magnitude (for example 8 bit values from 0 - 255) which set out the coordinate of a theoretical input image which maps to the corresponding "theoretical" output image having the same array coordinate indices. Unfortunately, any output image eg. 366 will have its own dimensions CxD which may further be totally different from an input image which may have its own dimensions ExF hence, it is necessary to facilitate the remapping of the warp map 365 so that it can be utilised for output image 366 to determine, for each output pixel, the corresponding area or region of the input image 367 from which the output pixel colour data is to be constructed. Hence, for each output pixel in output image 366 it is necessary to first determine a corresponding warp map value from warp map 365. This may include the need to buy

linearly interpolate the surrounding warp map values when an output image pixel maps to a fractional position within warp map Table 365. The result of this process will give the location of an input image pixel in a "theoretical" image which will be dimensioned by the size of each data value within the warp map 365. These values must be rescaled so as to map the theoretical image to the corresponding actual input image 367.

In order to determine the actual value and output image pixel should take so as to avoid aliasing effects, adjacent output image pixels should be examines to determine a region of input image pixels 367 which will contribute to the final output image pixel value. In this respect, the image pyramid is utilised as will become more apparent hereinafter.

The image warper performs several tasks in order to warp an image.

Scale the warp map to match the output image size.

Determine the span of the region of input image pixels represented in each output pixel.

Calculate the final output pixel value via tri-linear interpolation from the input image pyramid

Scale warp map

As noted previously, in a data driven warp, there is the need for a warp map that describes, for each output pixel, the center of a corresponding input image map. Instead of having a single warp map as previously described, containing interleaved x and y value information, it is possible to treat the X and Y coordinates as separate channels.

Consequently, preferably there are two warp maps: an X warp map showing the warping of X coordinates, and a Y warp map, showing the warping of the Y coordinate. As noted previously, the warp map 365 can have a different spatial resolution than the image they being scaled (for example a 32 x 32 warp-map 365 may adequately describe a warp for a

1500 x 1000 image 366). In addition, the warp maps can be represented by 8 or 16 bit values that correspond to the size of the image being warped.

There are several steps involved in producing points in the input image space from a given warp map:

1. Determining the corresponding position in the warp map for the output pixel
2. Fetch the values from the warp map for the next step (this can require scaling in the resolution domain if the warp map is only 8 bit values)
3. Bi-linear interpolation of the warp map to determine the actual value
4. Scaling the value to correspond to the input image domain

The first step can be accomplished by multiplying the current X/Y coordinate in the output image by a scale factor (which can be different in X & Y). For example, if the output image was 1500 x 1000, and the warp map was 150 x 100, we scale both X & Y by 1/10.

Fetching the values from the warp map requires access to 2 Lookup tables. One Lookup table indexes into the X warp-map, and the other indexes into the Y warp-map. The lookup table either reads 8 or 16 bit entries from the lookup table, but always returns 16 bit values (clearing the high 8 bits if the original values are only 8 bits).

The next step in the pipeline is to bi-linearly interpolate the looked-up warpmap values.

Finally the result from the bi-linear interpolation is scaled to place it in the same domain as the image to be warped. Thus, if the warp map range was 0-255, we scale X by 1500/255, and Y by 1000/255.

The interpolation process is as illustrated in Fig. 44 with the following constants set by software:

| Constant | Value |
|---|---|
| $K_1$ | Xscale (scales 0-ImageWidth to 0-WarpmapWidth) |
| $K_2$ | Yscale (scales 0-ImageHeight to 0-WarpmapHeight) |

| | |
|---|---|
| $K_3$ | XrangeScale (scales warpmap range (eg 0-255) to 0-ImageWidth) |
| $K_4$ | YrangeScale (scales warpmap range (eg 0-255) to 0-ImageHeight) |

The following lookup table is used:

| Lookup | Size | Details |
|---|---|---|
| $LU_1$ and $LU_2$ | WarpmapWidth x WarpmapHeight | Warpmap lookup. Given [X,Y] the 4 entries required for bi-linear interpolation are returned. Even if entries are only 8 bit, they are returned as 16 bit (high 8 bits 0). Transfer time is 4 entries at 2 bytes per entry. Total time is 8 cycles as 2 lookups are used. |

Span calculation

The points from the warp map 365 locate centers of pixel regions in the input image 367. The distance between input image pixels of adjacent output image pixels will indicate the size of the regions, and this distance can be approximated via a span calculation.

Turning to Fig.45, for a given current point in the warp map P1. The previous point on the same line is called P0, and the previous line's point at the same position is called P2. We determine the absolute distance in X & Y between P1 and P0, and between P1 and P2. The maximum distance in X or Y becomes the span which will be a square approximation of the actual shape.

Preferably, the points are processed in a vertical strip output order, P0 is the previous point on the same line within a strip, and when P1 is the first point on line within a strip, then P0 refers to the last point in the previous strip's corresponding line. P2 is the previous line's point in the same strip, so it can be kept in a 32-

entry history buffer. The basic of the calculate span process are as illustrated in Fig. 46 with the details of the process as illustrated in Fig. 47.

The following DRAM FIFO is used:

| Lookup | Size | Details |
|---|---|---|
| $FIFO_1$ | 8 ImageWidth bytes. [ImageWidth x 2 entries at 32 bits per entry] | P2 history/lookup (both X & Y in same FIFO) P1 is put into the FIFO and taken out again at the same pixel on the following row as P2. Transfer time is 4 cycles (2 x 32 bits, with 1 cycle per 16 bits) |

Since a 32 bit precision span history is kept, in the case of a 1500 pixel wide image being warped 12,000 bytes temporary storage is required.

Calculation of the span 364 uses 2 Adder ALUs (1 for span calculation, 1 for looping and counting for P0 and P2 histories) takes 7 cycles as follows:

| Cycle | Action |
|---|---|
| 1 | $A = ABS(P1_x - P2_x)$<br>Store $P1_x$ in $P2_x$ history |
| 2 | $B = ABS(P1_x - P0_x)$<br>Store $P1_x$ in $P0_x$ history |
| 3 | $A = MAX(A, B)$ |
| 4 | $B = ABS(P1_y - P2_y)$<br>Store $P1_y$ in $P2_y$ history |
| 5 | $A = MAX(A, B)$ |
| 6 | $B = ABS(P1_y - P0_y)$<br>Store $P1_y$ in $P0_y$ history |
| 7 | $A = MAX(A, B)$ |

The history buffers 365, 366 are cached DRAM. The 'Previous Line' (for P2 history) buffer 366 is 32 entries of span-precision. The 'Previous Point' (for P0 history). Buffer 365 requires 1 register that is used most of the time (for calculation of points 1 to 31 of a line in a strip), and a DRAM buffered set of history values to be used in the calculation of point 0 in a strip's line.

32 bit precision in span history requires 4 cache lines to hold P2 history, and 2 for P0 history. P0's history is only written and read out once every 8 lines of 32 pixels to a temporary storage space of (ImageHeight*4) bytes. Thus a 1500 pixel high image being warped requires 6000 bytes temporary storage, and a total of 6 cache lines.

Tri-linear interpolation

Having determined the center and span of the area from the input image to be averaged, the final part of the warp process is to determine the value of the output pixel. Since a single output pixel could theoretically be represented by the entire input image, it is potentially too time-consuming to actually read and average the specific area of the input image contributing to the output pixel. Instead, it is possible to approximate the pixel value by using an image pyramid of the input image.

If the span is 1 or less, it is necessary only to read the original image's pixels around the given coordinate, and perform bi-linear interpolation. If the span is greater than 1, we must read two appropriate levels of the image pyramid and perform tri-linear interpolation. Performing linear interpolation between two levels of the image pyramid is not strictly correct, but gives acceptable results (it errs on the side of blurring the resultant image).

Turning to Fig.48, generally speaking, for a given span 's', it is necessary to read image pyramid levels given by $ln_2s$ 370 and $ln_2s+1$ 371. $Ln_2s$ is simply decoding the highest set bit of s. We must bi-linear interpolate to determine the value for the pixel value on each of the two levels 370,371

of the pyramid, and then interpolate between

As shown in Fig.49, it is necessary to first interpolate in X and Y for each pyramid level before interpolating between the pyramid levels to obtain a final output value 373.

The image pyramid address mode issued to generate addresses for pixel coordinates at (x, y) on pyramid level s & s+1. Each level of the image pyramid contains pixels sequential in x. Hence, reads in x are likely to be cache hits.

Reasonable cache coherence can be obtained as local regions in the output image are typically locally coherent in the input image (perhaps at a different scale however, but coherent within the scale). Since it is not possible to know the relationship between the input and output images, we ensure that output pixels are written in a vertical strip (via a Vertical-Strip Iterator) in order to best make use of cache coherence.

Tri-linear interpolation can be completed in as few as 2 cycles on average using all 4 multiply ALUs and all 4 adder ALUs as a pipeline and assuming no memory access required. But since all the interpolation values are derived from the image pyramids, interpolation speed is completely dependent on cache coherence (not to mention the other units are busy doing warp-map scaling and span calculations). As many cache lines as possible should therefore be available to the image-pyramid reading. The best speed will be 8 cycles, using 2 Multiply ALUs (see the chapter on ALUs for a discussion on different algorithms for tri-linear interpolation).

The output pixels are written out to the DRAM via a Vertical-Strip Write Iterator that uses 2 cache lines. The speed is therefore limited to a minimum of 8 cycles per output pixel. If the scaling of the warp map requires 8 or fewer cycles, then the overall speed will be unchanged. Otherwise the throughput is the time taken to scale the warp

map. In most cases the warp map will be scaled up to match the size of the photo.

Assuming a warp map that requires 8 or fewer cycles per pixel to scale, the time taken to convert a single colour component of image is therefore 0.12s (1500 * 1000 * 8 cycles * 10ns per cycle).

Histogram Collector

The histogram collector is a microcode program that takes an image channel as input, and produces a histogram as output. Each of a channel's pixels has a value in the range 0-255. Consequently there are 256 entries in the histogram table, each entry 32 bits - large enough to contain a count of an entire 1500x1000 image.

As shown in Fig.50, since the histogram represents a summary of the entire image, a Sequential Read Iterator 378 is sufficient for the input. The histogram itself can be completely cached, requiring 32 cache lines (1K).

The microcode has two passes: an initialization pass which sets all the counts to zero, and then a "count" stage that increments the appropriate counter for each pixel read from the image.

The first stage requires the Address Unit and a single Adder ALU, with the address of the histogram table 377 for initializing.

| Relative Microcode Address | Address Unit<br>A = Base address of histogram | Adder Unit 1 |
|---|---|---|
| 0 | Write 0 to<br>A + (Adder1.Out1 << 2) | Out1 = A<br>A = A - 1<br>BNZ 0 |
| 1 | Rest of processing | Rest of processing |

The second stage processes the actual pixels from the image, and uses 4 Adder ALUs:

| | Adder 1 | Adder 2 | Adder 3 | Adder 4 | Address Unit |
|---|---|---|---|---|---|
| 1 | A = 0 | | | A = -1 | |
| 2<br>BZ<br>2 | Out1 = A<br>A = pixel | A = Adder1.Out1<br>Z = pixel - Adder1.Out1 | A = Adr.Out1 | A = A + 1 | Out1 = Read 4 bytes from: (A + (Adder1.Out1 << 2)) |
| 3 | | Out1 = A | Out1 = A | Out1 = A<br>A = Adder3.Out1 | Write Adder4.Out1 to: (A + (Adder 2.Out << 2) |
| 4 | | | | | Write Adder4.Out1 to: (A + (Adder 2.Out << 2)<br>Flush caches |

The Zero flag from Adder2 cycle 2 is used to stay at microcode address 2 for as long as the input pixel is the same. When it changes, the new count is written out in microcode address 3, and processing resumes at microcode address 2. Microcode address 4 is used at the end, when there are no more pixels to be read.

Stage 1 takes 256 cycles, or 2560ns. Stage 2 varies according to the values of the pixels. The worst case time for lookup table replacement is 2 cycles per image pixel if every pixel is not the same as its neighbor. The time taken for a single colour lookup is 0.03s (1500 x 1000 x 2 cycle per pixel x 10ns per cycle = 30,000,000ns). The time taken for 3 colour components is 3 times this amount, or 0.09s. There is no speed gain by combining the

Color Transform

Color transformation is achieved in two main ways:

Lookup table replacement

Color space conversion

Lookup Table Replacement

The input image is processed simultaneously in two halves to make effective use of memory bandwidth. The process is as indicated in Fig. 51 and

As illustrated in Fig.51, one of the simplest ways to transform the colour of a pixel is to encode an arbitrarily complex transform function into a lookup table 380. The component colour value of the pixel is used to lookup 381 the new component value of the pixel. For each pixel read from a Sequential Read Iterator, its new value is read from the New Colour Table 380, and written to a Sequential Write Iterator 383. The input image can be processed simultaneously in two halves to make effective use of memory bandwidth. The following lookup table is used:

| Lookup | Size | Details |
|---|---|---|
| $LU_1$ | 256 entries 8 bits per entry | Replacement[X] Table indexed by the 8 highest significant bits of X. Resultant 8 bits treated as fixed point 0:8 |

The total process requires 2 Sequential Read Iterators and 2 Sequential Write iterators. The 2 New Colour Tables require 8 cache lines each to hold the 256 bytes (256 entries of 1 byte).

The average time for lookup table replacement is therefore ½ cycle per image pixel. The time taken for a single colour lookup is 0.0075s (1500 x 1000 x ½ cycle per pixel x 10ns per cycle = 7,500,000ns). The time taken for 3 colour components is 3 times this amount, or 0.0225s. Each colour component has to be processed one after the other under control of software.

Colour Space Conversion

Colour Space conversion is only required when moving between colour spaces. The CCD images are captured in RGB colour space, and printing occurs in CMY colour space, while clients of the ACP 31 likely process images in the Lab

colour space. All of the input colour space channels are typically required as input to determine each output channel's component value. Thus the logical process is as illustrated 385 in Fig.52.

Simply, conversion between Lab, RGB, and CMY is fairly straightforward. However the individual colour profile of a particular device can vary considerably. Consequently, to allow future CCDs, inks, and printers, the ACP 31 performs colour space conversion by means of tri-linear interpolation from colour space conversion lookup tables.

Colour coherence tends to be area based rather than line based. To aid cache coherence during tri-linear interpolation lookups, it is best to process an image in vertical strips. Thus the read 386-388 and write 389 iterators would be Vertical-Strip Iterators.

Tri-linear colour space conversion

For each output colour component, a single 3D table mapping the input colour space to the output colour component is required. For example, to convert CCD images from RGB to Lab, 3 tables calibrated to the physical characteristics of the CCD are required:

RGB->L

RGB->a

RGB->b

To convert from Lab to CMY, 3 tables calibrated to the physical characteristics of the ink/printer are required:

Lab->C

Lab->M

Lab->Y

The 8-bit input colour components are treated as fixed-point numbers (3:5) in order to index into the conversion tables. The 3 bits of integer give the index, and the 5 bits of fraction are used for interpolation. Since 3 bits gives 8 values, 3 dimensions gives 512 entries (8 x 8 x 8). The size of each entry is 1 byte, requiring 512 bytes per table.

The Convert Color Space process can therefore be implemented as shown in Fig. 53 and the following lookup table is used:

| Lookup | Size | Details |
|---|---|---|
| $LU_1$ | 8 x 8 x 8 entries<br>512 entries<br>8 bits per entry | Convert[X, Y, Z]<br>Table indexed by the 3 highest bits of X, Y, and Z.<br>8 entries returned from Tri-linear index address unit<br>Resultant 8 bits treated as fixed point 8:0<br>Transfer time is 8 entries at 1 byte per entry |

Tri-linear interpolation returns interpolation between 8 values. Each 8 bit value takes 1 cycle to be returned from the lookup, for a total of 8 cycles. The tri-linear interpolation also takes 8 cycles when 2 Multiply ALUs are used per cycle. General tri-linear interpolation information is given in the ALU section of this document. The 512 bytes for the lookup table fits in 16 cache lines.

The time taken to convert a single colour component of image is therefore 0.105s (1500 * 1000 * 7 cycles * 10ns per cycle). To convert 3 components takes 0.415s. Fortunately the colour space conversion for printout takes place on the fly during printout itself, so is not a perceived delay.

If colour components are converted separately, they must not overwrite their input colour space components since all colour components from the input colour space are required for converting each component.

Since only 1 multiply unit is used to perform the interpolation, it is alternatively possible to do the entire Lab->CMY conversion as a single pass. This would require 3 Vertical-Strip Read Iterators, 3 Vertical-Strip Write Iterators, and access to 3 conversion tables simultaneously. In that case, it is possible to write back onto the input image and thus use no extra memory. However, access to 3

conversion tables equals 1/3 of the caching for each, that could lead to high latency for the overall process.

Affine Transform

Prior to compositing an image with a photo, it may be necessary to rotate, scale and translate it. If the image is only being translated, it can be faster to use a direct sub-pixel translation function. However, rotation, scale-up and translation can all be incorporated into a single affine transform.

A general affine transform can be included as an accelerated function. Affine transforms are limited to 2D, and if scaling down, input images should be pre-scaled via the Scale function. Having a general affine transform function allows an output image to be constructed one block at a time, and can reduce the time taken to perform a number of transformations on an image since all can be applied at the same time.

A transformation matrix needs to be supplied by the client - the matrix should be the inverse matrix of the transformation desired i.e. applying the matrix to the output pixel coordinate will give the input coordinate.

A 2D matrix is usually represented as a 3 x 3 array:

Since the 3rd column is always[0, 0, 1] clients do not need to specify it. Clients instead specify a, b, c, d, e, and f.

Given a coordinate in the output image (x, y) whose top left pixel coordinate is given as (0, 0), the input coordinate is specified by: (ax + cy + e, bx + dy + f). Once the input coordinate is determined, the input image is sampled to arrive at the pixel value. Bi-linear interpolation of input image pixels is used to determine the value of the pixel at the calculated coordinate.

Once the input coordinate is determined, the input image is sampled to arrive at the pixel value by bi-linear interpolation. Since affine transforms preserve parallel lines, images are processed in output vertical strips of 32

pixels wide for best average input image cache coherence.

3 Multiply ALUs are required to perform the bi-linear interpolation in 2 cycles. Multiply ALUs 1 and 2 do linear interpolation in X for lines Y and Y+1 respectively, and Multiply ALU 3 does linear interpolation in Y between the values output by Multiply ALUs 1 and 2.

As we move to the right across an output line in X, 2 Adder ALUs calculate the actual input image coordinates by adding 'a' to the current X value, and 'b' to the current Y value respectively. When we advance to the next line (either the next line in a vertical strip after processing a maximum of 32 pixels, or to the first line in a new vertical strip) we update X and Y to pre-calculated start coordinate values constants for the given block

The process for calculating an input coordinate is given in Fig. 54 where the following constants are set by software:

| Constant | Value |
|---|---|
| $K_1$ | c |
| $K_2$ | a |
| $K_3$ | e |
| $K_4$ | b |
| $K_5$ | d |
| $K_6$ | f |

Calculate Pixel

Once we have the input image coordinates, the input image must be sampled. A lookup table is used to return the values at the specified coordinates in readiness for bilinear interpolation. The basic process is as indicated in Fig. 55 and the following lookup table is used:

| Lookup | Size | Details |
|---|---|---|
| $LU_1$ | Image | Bilinear Image lookup [X, Y] |

| | | |
|---|---|---|
| | width by Image height 8 bits per entry | Table indexed by the integer part of X and Y.<br>4 entries returned from Bilinear index address unit, 2 per cycle.<br>Each 8 bit entry treated as fixed point 8:0<br>Transfer time is 2 cycles (2 16 bit entries in FIFO hold the 4 8 bit entries) |

The affine transform requires all 4 Multiply Units and all 4 Adder ALUs, and *with good cache coherence* can perform an affine transform with an average of 2 cycles per output pixel. This timing assumes good cache coherence, which is true for non-skewed images. Worst case timings are severely skewed images, which meaningful Vark scripts are unlikely to contain.

The time taken to transform a 128 x 128 image is therefore 0.00033 seconds (32,768 cycles). If this is a clip image with 4 channels (including a channel), the total time taken is 0.00131 seconds (131,072 cycles).

A Vertical-Strip Write Iterator is required to output the pixels. No Read Iterator is required. However, since the affine transform accelerator is bound by time taken to access input image pixels, as many cache lines as possible should be allocated to the read of pixels from the input image. At least 32 should be available, and preferably 64 or more.

Scaling

Scaling is essentially a re-sampling of an image. Scale *up* of an image can be performed using the Affine Transform function. Generalized scaling of an image, including scale down, is performed by the hardware accelerated Scale function. Scaling is performed independently in X and Y, so different scale factors can be used in each dimension.

The generalized scale unit must match the Affine Transform scale function in terms of registration. The generalized scaling process is as illustrated in Fig. 56.

The scale in X is accomplished by Fant's resampling algorithm as illustrated in Fig. 57.

Where the following constants are set by software:

| Constant | Value |
|---|---|
| $K_1$ | Number of input pixels that contribute to an output pixel in X |
| $K_2$ | $1/K_1$ |

The following registers are used to hold temporary variables:

| Variable | Value |
|---|---|
| $Latch_1$ | Amount of input pixel remaining unused (starts at 1 and decrements) |
| $Latch_2$ | Amount of input pixels remaining to contribute to current output pixel (starts at $K_1$ and decrements) |
| $Latch_3$ | Next pixel (in X) |
| $Latch_4$ | Current pixel |
| $Latch_5$ | Accumulator for output pixel (unscaled) |
| $Latch_6$ | Pixel Scaled in X (output) |

The Scale in Y process is illustrated in Fig. 58 and is also accomplished by a slightly altered version of Fant's resampling algorithm to account for processing in order of X pixels. The implementation is shown here:

Where the following constants are set by software:

| Constant | Value |
|---|---|
| $K_1$ | Number of input pixels that contribute to an output pixel in Y |
| $K_2$ | $1/K_1$ |

The following registers are used to hold temporary variables:

| Variable | Value |
|---|---|

| | |
|---|---|
| $Latch_1$ | Amount of input pixel remaining unused (starts at 1 and decrements) |
| $Latch_2$ | Amount of input pixels remaining to contribute to current output pixel (starts at $K_1$ and decrements) |
| $Latch_3$ | Next pixel (in Y) |
| $Latch_4$ | Current pixel |
| $Latch_5$ | Pixel Scaled in Y (output) |

The following DRAM FIFOs are used:

| Lookup | Size | Details |
|---|---|---|
| $FIFO_1$ | $ImageWidth_{OUT}$ entries<br>8 bits per entry | 1 row of image pixels already scaled in X<br>1 cycle transfer time |
| $FIFO_2$ | $ImageWidth_{OUT}$ entries<br>16 bits per entry | 1 row of image pixels already scaled in X<br>2 cycles transfer time (1 byte per cycle) |

Tessellate Image

Tessellation of an image is a form of tiling. It involves copying a specially designed "tile" multiple times horizontally and vertically into a second (usually larger) image space. When tessellated, the small tile forms a seamless picture. One example of this is a small tile of a section of a brick wall. It is designed so that when tessellated, it forms a full brick wall. Note that there is no scaling or sub-pixel translation involved in tessellation.

The most cache-coherent way to perform tessellation is to output the image sequentially line by line, and to repeat the same line of the input image for the duration of the line. When we finish the line, the input image must also advance to the next line (and repeat it multiple times across the output line).

An overview of the tessellation function is illustrated in Fig.59:

The Sequential Read Iterator 392 is set up to continuously read a single line of the input tile (StartLine would be 0 and EndLine would be 1). Each input pixel is written to all 3 of the Write Iterators 393-395. A counter 397 in an Adder ALU counts down the number of pixels in an output line, terminating the sequence at the end of the line.

At the end of processing a line, a small software routine updates the Sequential Read Iterator's StartLine & EndLine registers before restarting the microcode and the Sequential Read Iterator (which clears the FIFO and repeats line 2 of the tile). The Write Iterators 393-395 are not updated, and simply keep on writing out to their respective parts of the output image.

The net effect is that the tile has one line repeated across an output line, and then the tile is repeated vertically too.

This process does not fully use the memory bandwidth since we get good cache coherence in the input image, but it does allow the tessellation to function with tiles of any size. The process uses 1 Adder ALU. If the 3 Write Iterators 393-395 each write to 1/3 of the image (breaking the image on tile sized boundaries), then the entire tessellation process takes place at an average speed of 1/3 cycle per output image pixel. For an image of 1500 x 1000, this equates to .005 seconds (5,000,000ns).

Sub-pixel Translator

Before compositing an image with a background, it may be necessary to translate it by a sub-pixel amount in both X and Y. Sub-pixel transforms can increase an image's size by 1 pixel in each dimension. The value of the region outside the image can be client determined, such as a constant value (e.g. black), or edge pixel replication. Typically it will be better to use black.

The sub-pixel translation process is as illustrated in Fig. 60. Sub-pixel translation in a given dimension is defined by:

$Pixel_{out} = Pixel_{in} * (1-Translation) + Pixel_{in-1} * Translation$

It can also be represented as a form of interpolation:

$Pixel_{out} = Pixel_{in-1} + (Pixel_{in} - Pixel_{in-1}) * Translation$

Implementation of a single (on average) cycle interpolation engine using a single Multiply ALU and a single Adder ALU in conjunction is straightforward. Sub-pixel translation in both X & Y requires 2 interpolation engines.

In order to sub-pixel translate in Y, 2 Sequential Read Iterators 400, 401 are required (one is reading a line ahead of the other from the same image), and a single Sequential Write Iterator 403 is required.

The first interpolation engine (interpolation in Y) accepts pairs of data from 2 streams, and linearly interpolates between them. The second interpolation engine (interpolation in X) accepts its data as a single 1 dimensional stream and linearly interpolates between values. Both engines interpolate in 1 cycle on average. Descriptions of interpolators and example microcode for the engines can be found in the ALU section of this document.

Each interpolation engine 405, 406 is capable of performing the sub-pixel translation in 1 cycle per output pixel on average. The overall time is therefore 1 cycle per output pixel, with requirements of 2 Multiply ALUs and 2 Adder ALUs.

The time taken to output 32 pixels from the sub-pixel translate function is on average 320ns (32 cycles). This is enough time for 4 full cache-line accesses to DRAM, so the use of 3 Sequential Iterators is well within timing limits.

The total time taken to sub-pixel translate an image is therefore 1 cycle per pixel of the output image. A typical image to be sub-pixel translated is a tile of size 128 *

128. The output image size is 129 * 129. The process takes 129 * 129 * 10ns = 166,410ns.

The Image Tiler function also makes use of the sub-pixel translation algorithm, but does not require the writing out of the sub-pixel-translated data, but rather processes it further.

Image Tiler

The high level algorithm for tiling an image is carried out in software. Once the placement of the tile has been determined, the appropriate coloured tile must be composited. The actual compositing of each tile onto an image is carried out in hardware via the microcoded ALUs. Compositing a tile involves both a texture application and a colour application to a background image. In some cases it is desirable to compare the *actual* amount of texture added to the background in relation to the *intended* amount of texture, and use this to scale the colour being applied. In these cases the texture must be applied first.

Since colour application functionality and texture application functionality are somewhat independent, they are separated into sub-functions.

The number of cycles per 4-channel tile composite for the different texture styles and colouring styles is summarized in the following table:

| | Constant colour | Pixel colour |
|---|---|---|
| Replace texture | 4 | 4.75 |
| 25% background + tile texture | 4 | 4.75 |
| Average height algorithm | 5 | 5.75 |
| Average height algorithm with feedback | 5.75 | 6.5 |

Tile Colouring and Compositing

A tile is set to have either a constant colour (for the whole tile), or takes each pixel value from an input image.

Both of these cases may also have feedback from a texturing stage to scale the opacity (similar to thinning paint).

The steps for the 4 cases can be summarized as:

Sub-pixel translate the tile's opacity values,

Optionally scale the tile's opacity (if feedback from texture application is enabled).

Determine the colour of the pixel (constant or from an image map).

Composite the pixel onto the background image.

Each of the 4 cases is treated separately, in order to minimize the time taken to perform the function. The summary of time per colour compositing style for a single colour channel is described in the following table:

| Tiling color style | No feedback from texture (cycles per pixel) | Feedback from texture (cycles per pixel) |
|---|---|---|
| Tile has constant color per pixel | 1 | 2 |
| Tile has per pixel color from input image | 1.25 | 2 |

Constant colour

In this case, the tile has a constant colour, determined by software. While the ACP 31 is placing down one tile, the software can be determining the placement and colouring of the next tile.

The colour of the tile can be determined by bi-linear interpolation into a scaled version of the image being tiled. The scaled version of the image can be created and stored in place of the image pyramid, and needs only to be performed once per entire tile operation. If the tile size is 128 x 128, then the image can be scaled down by 128:1 in

each dimension.

Without feedback

When there is no feedback from the texturing of a tile, the tile is simply placed at the specified coordinates. The tile colour is used for each pixel's colour, and the opacity for the composite comes from the tile's sub-pixel translated opacity channel. In this case colour channels and the texture channel can be processed completely independently between tiling passes.

The overview of the process is illustrated in Fig.61. Sub-pixel translation 410 of a tile can be accomplished using 2 Multiply ALUs and 2 Adder ALUs in an average time of 1 cycle per output pixel. The output from the sub-pixel translation is the mask to be used in compositing 411 the constant tile colour 412 with the background image from background sequential Read Iterator 41.

Compositing can be performed using 1 Multiply ALU and 1 Adder ALU in an average time of 1 cycle per composite.

Requirements are therefore 3 Multiply ALUs and 3 Adder ALUs. 4 Sequential Iterators 413-416 are required, taking 320ns to read or write their contents. With an average number of cycles of 1 per pixel to sub-pixel translate and composite, there is sufficient time to read and write the buffers.

With feedback

When there is feedback from the texturing of a tile, the tile is placed at the specified coordinates. The tile colour is used for each pixel's colour, and the opacity for the composite comes from the tile's sub-pixel translated opacity channel scaled by the feedback parameter. Thus the texture values must be calculated before the colour value is applied.

The overview of the process is illustrated in Fig.62. Sub-pixel translation of a tile can be accomplished using 2 Multiply ALUs and 2 Adder ALUs in an average time of 1 cycle per output pixel. The output from the sub-pixel translation

is the mask to be scaled according to the feedback read from the Feedback Sequential Read Iterator 420. The feedback is passed it to a Scaler (1 Multiply ALU) 421.

Compositing 422 can be performed using 1 Multiply ALU and 1 Adder ALU in an average time of 1 cycle per composite.

Requirements are therefore all 4 Multiply ALUs and all 4 Adder ALUs. Although the entire process can be accomplished in 1 cycle on average, the bottleneck is the memory access, since 5 Sequential Iterators are required. With sufficient buffering, the average time is 1.25 cycles per pixel.

Colour from Input Image

One way of colouring pixels in a tile is to take the colour from pixels in an input image. Again, there are two possibilities for compositing: with and without feedback from the texturing.

Without feedback

In this case, the tile colour simply comes from the relative pixel in the input image. The opacity for compositing comes from the tile's opacity channel sub-pixel shifted.

The overview of the process is illustrated in Fig.63. Sub-pixel translation 425 of a tile can be accomplished using 2 Multiply ALUs and 2 Adder ALUs in an average time of 1 cycle per output pixel. The output from the sub-pixel translation is the mask to be used in compositing 426 the tile's pixel colour (read from the input image 428 ) with the background image 429.

Compositing 426 can be performed using 1 Multiply ALU and 1 Adder ALU in an average time of 1 cycle per composite.

Requirements are therefore 3 Multiply ALUs and 3 Adder ALUs. Although the entire process can be accomplished in 1 cycle on average, the bottleneck is the memory access, since 5 Sequential Iterators are required. With sufficient buffering, the average time is 1.25 cycles per pixel.

With feedback

In this case, the tile colour still comes from the relative pixel in the input image, but the opacity for compositing is affected by the relative amount of texture height actually applied during the texturing pass. This process is as illustrated in Fig. 64.

Sub-pixel translation 431 of a tile can be accomplished using 2 Multiply ALUs and 2 Adder ALUs in an average time of 1 cycle per output pixel. The output from the sub-pixel translation is the mask to be scaled 431 according to the feedback read from the Feedback Sequential Read Iterator 432. The feedback is passed to a Scaler (1 Multiply ALU) 431.

Compositing 434 can be performed using 1 Multiply ALU and 1 Adder ALU in an average time of 1 cycle per composite.

Requirements are therefore all 4 Multiply ALUs and 3 Adder ALUs. Although the entire process can be accomplished in 1 cycle on average, the bottleneck is the memory access, since 6 Sequential Iterators are required. With sufficient buffering, the average time is 1.5 cycles per pixel.

Tile Texturing

Each tile has a surface texture defined by its texture channel. The texture must be sub-pixel translated and then applied to the output image. There are 3 styles of texture compositing:

Replace texture

25% background + tile's texture

Average height algorithm

In addition, the Average height algorithm can save feedback parameters for colour compositing.

The time taken per texture compositing style is summarized in the following table:

| Tiling colour style | Cycles per pixel (no | Cycles per pixel (feedbac |
|---|---|---|

| | feedback from texture) | k from texture) |
|---|---|---|
| Replace texture | 1 | - |
| 25% background + tile texture value | 1 | - |
| Average height algorithm | 2 | 2 |

Replace texture

In this instance, the texture from the tile replaces the texture channel of the image, as illustrated in Fig.65. Sub-pixel translation 436 of a tile's texture can be accomplished using 2 Multiply ALUs and 2 Adder ALUs in an average time of 1 cycle per output pixel. The output from this sub-pixel translation is fed directly to the Sequential Write Iterator 437.

The time taken for replace texture compositing is 1 cycle per pixel. Note that there is no feedback, since 100% of the texture value is always applied to the background. There is therefore no requirement for processing the channels in any particular order.

25% Background + Tile's Texture

In this instance, the texture from the tile is added to 25% of the existing texture value. The new value must be greater than or equal to the original value. In addition, the new texture value must be clipped at 255 since the texture channel is only 8 bits. The process utilised is illustrated in Fig.66.

Sub-pixel translation 440 of a tile's texture can be accomplished using 2 Multiply ALUs and 2 Adder ALUs in an average time of 1 cycle per output pixel. The output from this sub-pixel translation 440 is fed to an adder 441 where it is added to ¼ 442 of the background texture value. Min and Max functions 444 are provided by the 2 adders not used for sub-pixel translation and the output written to a Sequential Write Iterator 445.

The time taken for this style of texture compositing is 1 cycle per pixel. There is no feedback, since 100% of the texture value is considered to have been applied to the background (even if clipping at 255 occurred). There is therefore no requirement for processing the channels in any particular order.

Average height algorithm

In this texture application algorithm, the average height under the tile is computed, and each pixel's height is compared to the average height. If the pixel's height is less than the average, the stroke height is added to the background height. If the pixel's height is greater than or equal to the average, then the stroke height is added to the average height. Thus background peaks thin the stroke. The height is constrained to increase by a minimum amount to prevent the background from thinning the stroke application to 0 (the minimum amount can be 0 however). The height is also clipped at 255 due to the 8-bit resolution of the texture channel.

There can be feedback of the difference in texture applied versus the expected amount applied. The feedback amount can be used as a scale factor in the application of the tile's colour.

In both cases, the average texture is provided by software, calculated by performing a bi-level interpolation on a scaled version of the texture map. Software would determine the next tile's average texture height while the current tile is being applied. Software must also provide the minimum thickness for addition, which is typically constant for the entire tiling process.

Without feedback

With no feedback, the texture is simply applied to the background texture, as shown in Fig.67.

4 Sequential Iterators are required, which means that if the process can be pipelined for 1 cycle, the memory is fast enough to keep up.

Sub-pixel translation 450 of a tile's texture can be accomplished using 2 Multiply ALUs and 2 Adder ALUs in an average time of 1 cycle per output pixel. Each Min & Max function 451,452 requires a separate Adder ALU in order to complete the entire operation in 1 cycle. Since 2 are already used by the sub-pixel translation of the texture, there are not enough remaining for a 1 cycle average time.

The average time for processing 1 pixel's texture is therefore 2 cycles. Note that there is no feedback, and hence the colour channel order of compositing is irrelevant.

With feedback

This is conceptually the same as the case without feedback, except that in addition to the standard processing of the texture application algorithm, it is necessary to also record the proportion of the texture actually applied. The proportion can be used as a scale factor for subsequent compositing of the tile's colour onto the background image. A flow diagram is illustrated in Fig.68.

The following lookup table is used:

| Lookup | Size | Details |
|---|---|---|
| $LU_1$ | 256 entries<br>16 bits per entry | 1/N<br>Table indexed by N (range 0-255)<br>Resultant 16 bits treated as fixed point 0:16 |

Each of the 256 entries in the software provided 1/N table 460 is 16 bits, thus requiring 16 cache lines to hold continuously.

Sub-pixel translation 461 of a tile's texture can be accomplished using 2 Multiply ALUs and 2 Adder ALUs in an average time of 1 cycle per output pixel. Each Min 462 & Max 463 function requires a separate Adder ALU in order to complete the entire operation in 1 cycle. Since 2 are already used by the sub-pixel translation of the texture, there are not enough remaining for a 1 cycle average time.

The average time for processing 1 pixel's texture is therefore 2 cycles. Sufficient space must be allocated for

the feedback data area (a tile sized image channel). The texture must be applied before the tile's colour is applied, since the feedback is used in scaling the tile's opacity.

CCD Image Interpolator

Images obtained from the CCD via the ISI 83 (Fig.3) are 750 x 500 pixels. When the image is captured via the ISI, the orientation of the camera is used to rotate the pixels by 0, 90, 180, or 270 degrees so that the top of the image corresponds to 'up'. Since every pixel only has an R, G, or B colour component (rather than all 3), the fact that these have been rotated must be taken into account when interpreting the pixel values. Depending on the orientation of the camera, each 2x2 pixel block has one of the configurations illustrated in Fig.69:

Several processes need to be performed on the CCD captured image in order to transform it into a useful form for processing:

Up-interpolation of low-sample rate colour components in CCD image (interpreting correct orientation of pixels)

Colour conversion from RGB to the internal colour space

Scaling of the internal space image from 750 x 500 to 1500 x 1000.

Writing out the image in a planar format

The entire channel of an image is required to be available at the same time in order to allow warping. In a low memory model (8MB), there is only enough space to hold a single channel at full resolution as a temporary object. Thus the colour conversion is to a single colour channel. The limiting factor on the process is the colour conversion, as it involves tri-linear interpolation from RGB to the internal colour space, a process that takes 0.026ns per channel (750 x 500 x 7 cycles per pixel x 10ns per cycle = 26,250,000ns).

It is important to perform the colour conversion *before* scaling of the internal colour space image as this reduces the number of pixels scaled (and hence the overall process

time) by a factor of 4.

The requirements for all of the transformations do not fit in the ALU scheme. The transformations are therefore broken into two phases:

Phase 1: Up-interpolation of low-sample rate colour components in CCD image (interpreting correct orientation of pixels)

Colour conversion from RGB to the internal colour space
Writing out the image in a planar format

Phase 2: Scaling of the internal space image from 750 x 500 to 1500 x 1000

Separating out the scale function implies that the small colour converted image must be in memory at the same time as the large one. The output from Phase 1 (0.5 MB) can be safely written to the memory area usually kept for the image pyramid (1 MB). The output from Phase 2 can be the general expanded CCD image. Separation of the scaling also allows the scaling to be accomplished by the Affine Transform, and also allows for a different CCD resolution that may not be a simple 1:2 expansion.

Phase 1: Up-interpolation of low-sample rate colour components.

Each of the 3 colour components (R, G, and B) needs to be up interpolated in order for colour conversion to take place for a given pixel. We have 7 cycles to perform the interpolation per pixel since the colour conversion takes 7 cycles.

Interpolation of G is straightforward and is illustrated in Fig.53. Depending on orientation, the actual pixel value G alternates between odd pixels on odd lines & even pixels on even lines, and odd pixels on even lines & even pixels on odd lines. In both cases, linear interpolation is all that is required. Interpolation of R and B components as illustrated in Fig.71 and 72, is more complicated, since in the horizontal and vertical directions As can be seen from the diagrams, access to 3 rows of pixels

simultaneously is required, so 3 Sequential Read Iterators are required, each one offset by a single row. In addition, we have access to the previous pixel on the same row via a latch for each row.

Each pixel therefore contains one component from the CCD, and the other 2 up-interpolated. When one component is being bi-linearly interpolated, the other is being linearly interpolated. Since the interpolation factor is a constant 0.5, interpolation can be calculated by an add and a shift 1 bit right (in 1 cycle), and bi-linear interpolation of factor 0.5 can be calculated by 3 adds and a shift 2 bits right (3 cycles). The total number of cycles required is therefore 4, using a single multiply ALU.

Fig. 73 illustrates the case for rotation 0 even line even pixel (EL, EP), and odd line odd pixel (OL, OP) and Fig. 74 illustrates the case for rotation 0 even line odd pixel (EL, OP), and odd line even pixel (OL, EP). The other rotations are simply different forms of these two expressions.

Color conversion

Color space conversion from RGB to Lab is achieved using the same method as that described in the general Color Space Convert function, a process that takes 8 cycles per pixel. Phase 1 processing can be described with reference to Fig. 75.

The up-interpolate of the RGB takes 4 cycles (1 Multiply ALU), but the conversion of the color space takes 8 cycles per pixel (2 Multiply ALUs) due to the lookup transfer time.

Phase 2

Scaling the image

This phase is concerned with up-interpolating the image from the CCD resolution (750 x 500) to the working photo resolution (1500 x 1000). Scaling is accomplished by running the Affine transform with a scale of 1:2. The timing of a

general affine transform is 2 cycles per output pixel, which in this case means an elapsed scaling time of 0.03 seconds.

Timing Summary

Illuminate Image

Once an image has been processed, it can be illuminated by one or more light sources.Light sources can be:

1. Directional - is infinitely distant so it casts parallel light in a single direction
2. Omni - casts unfocused lights in all directions.
3. Spot - casts a focused beam of light at a specific target point. There is a cone and penumbra associated with a spotlight.

The scene may also have an associated bump-map to cause reflection angles to vary. Ambient light is also optionally present in an illuminated scene.

In this description of accelerated illumination, we are concerned with illuminating one image channel by a single light source. Multiple light sources can be applied to a single image channel as multiple passes: one pass per light source. Multiple channels can be processed one at a time with or without a bump-map.

The viewing vector V is always perpendicular to the image plane.

The normal surface vector (N) at a pixel is computed from the bump-map if present. The default normal vector, in the absence of a bump-map, is perpendicular to the image plane i.e. N = [0, 0, 1].

The viewing vector V is always perpendicular to the image plane i.e. V = [0, 0, 1].

For a directional light source, the light source vector (L) from a pixel to the light source is constant across the entire image, so is computed once for the entire image. For an omni light source (at a finite distance), the light source vector is computed independently for each pixel.

A pixel's reflection of ambient light is computed according to: $I_a k_a O_d$

A pixel's diffuse and specular reflection of a light source is computed according to the Phong model:

$$f_{att} I_p [k_d O_d (N \bullet L) + k_s O_s (R \bullet V)^n]$$

When the light source is at infinity, the light source intensity is constant across the image.

Each light source has three contributions per pixel
Diffuse contribution
Specular contribution

The light source can be defined using the following variables:

| | |
|---|---|
| $d_L$ | Distance from light source |
| $f_{att}$ | Attenuation with distance [$f_{att} = 1 / d_L^2$] |
| R | Normalized reflection vector [$R = 2N(N \bullet L) - L$] |
| $I_a$ | Ambient light intensity |
| $I_p$ | Diffuse light coefficient |
| $k_a$ | Ambient reflection coefficient |
| $k_d$ | Diffuse reflection coefficient |
| $k_s$ | Specular reflection coefficient |
| $k_{sc}$ | Specular color coefficient |
| L | Normalized light source vector |
| N | Normalized surface normal vector |
| n | Specular exponent |
| $O_d$ | Object's diffuse color (i.e. image pixel color) |
| $O_s$ | Object's specular color ($k_{sc} O_d + (1 - k_{sc}) I_p$) |
| V | Normalized viewing vector [V = [0, 0, 1]] |

The same reflection coefficients ($k_a$, $k_s$, $k_d$) are used for each colour component.

A given pixel's value will be equal to the ambient contribution plus the sum of each light's diffuse and specular contribution.

Sub-Processes of Illumination Calculation

In order to calculate diffuse and specular contributions, a variety of other calculations are required. These are calculations of:

- $1/\sqrt{X}$
- N
- L
- N•L
- R•V
- $f_{att}$
- $f_{cp}$

Sub-processes are also defined for calculating the contributions of:

- ambient
- diffuse
- specular

The sub-processes can then be used to calculate the overall illumination of a light source. Since there are only 4 multiply ALUs, the microcode for a particular type of light source can have sub-processes intermingled appropriately for performance.

Calculation of $1/\sqrt{X}$

The Vark lighting model uses vectors. In many cases it is important to calculate the inverse of the length of the vector for normalization purposes. Calculating the inverse of the length requires the calculation of 1/SquareRoot[X].

Logically, the process can be represented as a process with inputs and outputs as shown in Fig.76. Referring to Fig. 77, the calculation can be made via a lookup of the estimation, followed by a single iteration of the following function:

$$V_{n+1} = \tfrac{1}{2} V_n(3 - XV_n^2)$$

The number of iterations depends on the accuracy

required. In this case only 16 bits of precision are required. The table can therefore have 8 bits of precision, and only a single iteration is necessary. The following constant is set by software:

| Constant | Value |
|---|---|
| $K_1$ | 3 |

The following lookup table is used:

| Lookup | Size | Details |
|---|---|---|
| $LU_1$ | 256 entries 8 bits per entry | 1/SquareRoot[X] Table indexed by the 8 highest significant bits of X. Resultant 8 bits treated as fixed point 0:8 |

Overview of Illumination Calculation

Calculation of N

N is the surface normal vector. When there is no bump-map, N is constant. When a bump-map is present, N must be calculated for each pixel.

No bump-map

When there is no bump-map, there is a fixed normal N that has the following properties:

$$N = [X_N, Y_N, Z_N] = [0, 0, 1]$$

$$||N|| = 1$$

$$1/||N|| = 1$$

$$\text{normalized } N = N$$

These properties can be used instead of specifically calculating the normal vector and $1/||N||$ and thus optimize other calculations.

With bump-map

As illustrated in Fig. 78, when a bump-map is present, N is calculated by comparing bump-map values in X and Y dimensions. The following diagram shows the calculationof N for pixel P1 in terms of the pixels in the same row and column, but not including the value at P1 itself. The

calculation of N is made resolution independent by multiplying by a scale factor (same scale factor in X & Y). This process can be represented as a process having inputs and outputs ($Z_N$ is always 1) as illustrated in Fig.79.

As $Z_N$ is always 1. Consequently $X_N$ and $Y_N$ *are not normalized yet* (since $Z_N = 1$). Normalization of N is delayed until after calculation of N•L so that there is only 1 multiply by 1/||N|| instead of 3.

An actual process for calculating N is illustrated in Fig.80.

The following constant is set by software:

| Constant | Value |
|---|---|
| $K_1$ | ScaleFactor (to make N resolution independent) |

Calculation of L

Directional lights

When a light source is infinitely distant, it has an effective constant light vector L. L is normalized and calculated by software such that:

$L = [X_L, Y_L, Z_L]$

$||L|| = 1$

$1/||L|| = 1$

These properties can be used instead of specifically calculating the L and 1/||L|| and thus optimize other calculations. This process is as illustrated in Fig. 81.

Omni lights and Spotlights

When the light source is notinfinitely distant, L is the vector from the current point P to the light source PL. Since $P = [X_P, Y_P, 0]$, L is given by:

$L = [X_L, Y_L, Z_L]$

$X_L = X_P - X_{PL}$

$Y_L = Y_P - Y_{PL}$

$Z_L = -Z_{PL}$

We normalize $X_L$, $Y_L$ and $Z_L$ by multiplying each by 1/||L||. The calculation of 1/||L|| (for later use in normalizing) is

accomplished by calculating

$$V = X_L^2 + Y_L^2 + Z_L^2$$

and then calculating $V^{-1/2}$

In this case, the calculation of L can be represented as a process with the inputs and outputs as indicated in Fig. 82.

$X_P$ and $Y_P$ are the coordinates of the pixel whose illumination is being calculated. $Z_P$ is always 0.

The actual process for calculating L can be as set out in Fig.83.

Where the following constants are set by software:

| Constant | Value |
|---|---|
| $K_1$ | $X_{PL}$ |
| $K_2$ | $Y_{PL}$ |
| $K_3$ | $Z_{PL}^2$ (as $Z_P$ is 0) |
| $K_4$ | $-Z_{PL}$ |

Calculation of N•L

Calculating the dot product of vectors N and L is defined as:

$$X_N X_L + Y_N Y_L + Z_N Z_L$$

No bump-map

When there is no bump-map N is a constant [0, 0, 1]. N•L therefore reduces to $Z_L$.

With bump-map

When there is a bump-map, we must calculate the dot product directly. Rather than take in normalized N components, we normalize after taking the dot product of a non-normalized N to a normalized L. L is either normalized by software (if it is constant), or by the Calculate L process. This process is as illustrated in Fig. 84.

Note that $Z_N$ is not required as input since it is defined to be 1. However $1/||N||$ is required instead, in

order to normalize the result. One actual process for calculating N•L is as illustrated in Fig. 85.

Calculation of R•V

R•V is required as input to specular contribution calculations. Since V = [0, 0, 1], only the Z components are required. R•V therefore reduces to:

$$R \bullet V = 2Z_N(N \bullet L) - Z_L$$

In addition, since the un-normalized $Z_N = 1$, normalized $Z_N = 1/||N||$

No bump-map

The simplest implementation is when N is constant (i.e. no bump-map). Since N and V are constant, N•L and R•V can be simplified:

$V = [0, 0, 1]$

$N = [0, 0, 1]$

$L = [X_L, Y_L, Z_L]$

$N \bullet L = Z_L$

$R \bullet V = 2Z_N(N \bullet L) - Z_L$

$= 2Z_L - Z_L$

$= Z_L$

When L is constant (Directional light source), a normalized $Z_L$ can be supplied by software in the form of a constant whenever R•V is required. When L varies (Omni lights and Spotlights), normalized $Z_L$ must be calculated on the fly. It is obtained as output from the Calculate L process.

With bump-map

When N is not constant, the process of calculating R•V is simply an implementation of the generalized formula:

$$R \bullet V = 2Z_N(N \bullet L) - Z_L$$

The inputs and outputs are as shown in Fig. 86 with the an actual implementation as shown in Fig. 87.

Calculation of Attenuation Factor

Directional lights

When a light source is infinitely distant, the intensity of the light does not vary across the image. The attenuation factor $f_{att}$ is therefore 1. This constant can be used to optimize illumination calculations for infinitely distant light sources.

Omni lights and Spotlights

When a light source is not infinitely distant, the intensity of the light can vary according to the following formula:

$f_{att} = f_0 + f_1/d + f_2/d^2$

Appropriate settings of coefficients $f_0$, $f_1$, and $f_2$ allow light intensity to be attenuated by a constant, linearly with distance, or by the square of the distance.

Since $d = ||L||$, the calculation of $f_{att}$ can be represented as a process with the following inputs and outputs as illustrated in Fig.88.

The actual process for calculating $f_{att}$ can be defined in Fig.89.

Where the following constants are set by software:

| Constant | Value |
|---|---|
| $K_1$ | $F_2$ |
| $K_2$ | $f_1$ |
| $K_3$ | $F_0$ |

Calculation of Cone and Penumbra Factor

Directional lights and Omni lights

These two light sources are not focused, and therefore have no cone or penumbra. The cone-penumbra scaling factor $f_{cp}$ is therefore 1. This constant can be used to optimize illumination calculations for Directional and Omni light sources.

Spotlights

A spotlight focuses on a particular target point (PT). The intensity of the Spotlight varies according to whether the particular point of the image is in the cone, in the

penumbra, or outside the cone/penumbra region.

Turning now to Fig.90, there is illustrated a graph of fcp with respect to the penumbra position. Inside the cone 470, $f_{cp}$ is 1, outside 471 the penumbra $f_{cp}$ is 0. From the edge of the cone through to the end of the penumbra, the light intensity varies according to a cubic function 472.

The various vectors for penumbra 475 and cone 476 calculation are as illustrated in Fig.90 and 91.

Looking at the surface of the image in 1 dimension as shown in Fig.91, 3 angles A, B, and C are defined. A is the angle between the target point 479, the light source 478, and the end of the cone 480. C is the angle between the target point 479, light source 478, and the end of the penumbra 481. Both are fixed for a given light source. B is the angle between the target point 479, the light source 478, and the position being calculated 482, and therefore changes with every point being calculated on the image.

We normalize the range A to C to be 0 to 1, and find the distance that B is along that angle range by the formula:

(B-A) / (C-A)

The range is forced to be in the range 0 to 1 by truncation, and this value used as a lookup for the cubic approximation of $f_{cp}$.

The calculation of $f_{att}$ can therefore be represented as a process with the inputs and outputs as illustrated in Fig. 93 with an actual process for calculating $f_{cp}$ is as shown in Fig.94 where the following constants are set by software:

| Constant | Value |
|---|---|
| $K_1$ | $X_{LT}$ |
| $K_2$ | $Y_{LT}$ |
| $K_3$ | $Z_{LT}$ |
| $K_4$ | A |
| $K_5$ | 1/(C-A). [MAXNUM if no penumbra] |

The following lookup tables are used:

| Lookup | Size | Details |
|---|---|---|
| $LU_1$ | 64 entries 16 bits per entry | Arcos(X)<br>Units are same as for constants $K_5$ and $K_6$<br>Table indexed by highest 6 bits<br>Result by linear interpolation of 2 entries<br>Timing is 2 * 8 bits * 2 entries = 4 cycles |
| $LU_2$ | 64 entries 16 bits per entry | Light Response function $f_{cp}$<br>F(1) = 0, F(0) = 1, others are according to cubic<br>Table indexed by 6 bits (1:5)<br>Result by linear interpolation of 2 entries<br>Timing is 2 * 8 bits = 4 cycles |

Calculation of Ambient Contribution

Regardless of the number of lights being applied to an image, the ambient light contribution is performed *once* for each pixel, and does not depend on the bump-map.

The ambient calculation process can be represented as a process with the inputs and outputs as illustrated in Fig.95. The implementation of the process requires multiplying each pixel from the input image ($O_d$) by a constant value ($I_a k_a$), as shown in Fig.96 where the following constant is set by software:

| Constant | Value |
|---|---|
| $K_1$ | $I_a k_a$ |

Calculation of Diffuse Contribution

Each light that is applied to a surface produces a diffuse illumination. The diffuse illumination is given by

the formula:

$$\text{diffuse} = k_d O_d (N \bullet L)$$

There are 2 different implementations to consider:

Implementation 1 - constant N and L

When N and L are both constant (Directional light and no bump-map):

$$N \bullet L = Z_L$$

Therefore:

$$\text{diffuse} = k_d O_d Z_L$$

Since $O_d$ is the only variable, the actual process for calculating the diffuse contribution is as illustrated in Fig. 97 where the following constant is set by software:

| Constant | Value |
|---|---|
| $K_1$ | $k_d(N \bullet L) = k_d Z_L$ |

Implementation 2 - non-constant N & L

When either N or L are non-constant (either a bump-map or illumination from an Omni light or a Spotlight), the diffuse calculation is performed directly according to the formula:

$$\text{diffuse} = k_d O_d (N \bullet L)$$

The diffuse calculation process can be represented as a process with the inputs as illustrated in Fig. 98. N•L can either be calculated using the Calculate N•L Process, or is provided as a constant. An actual process for calculating the diffuse contribution is as shown in Fig. 99 where the following constants are set by software:

| Constant | Value |
|---|---|
| $K_1$ | $k_d$ |

Calculation of Specular Contribution

Each light that is applied to a surface produces a specular illumination. The specular illumination is given by the formula:

$$specular = k_s O_s (R \bullet V)^n$$

where $O_s = k_{sc} O_d + (1-k_{sc}) I_p$

There are two implementations of the Calculate Specular process.

Implementation 1 - constant N and L

The first implementation is when both N and L are constant (Directional light and no bump-map). Since N, L and V are constant, N•L and R•V are also constant:

$V = [0, 0, 1]$

$N = [0, 0, 1]$

$L = [X_L, Y_L, Z_L]$

$N \bullet L = Z_L$

$R \bullet V = 2Z_N (N \bullet L) - Z_L$

$= 2Z_L - Z_L$

$= Z_L$

The specular calculation can thus be reduced to:

$specular = k_s O_s Z_L^n$

$= k_s Z_L^n (k_{sc} O_d + (1-k_{sc}) I_p)$

$= k_s k_{sc} Z_L^n O_d + (1-k_{sc}) I_p k_s Z_L^n$

Since only $O_d$ is a variable in the specular calculation, the calculation of the specular contribution can therefore be represented as a process with the inputs and outputs as indicated in Fig. 100 and an actual process for calculating the specular contribution is illustrated in Fig. 101 where the following constants are set by software:

| Constant | Value |
|---|---|
| $K_1$ | $k_s k_{sc} Z_L^n$ |
| $K_2$ | $(1-k_{sc}) I_p k_s Z_L^n$ |

Implementation 2 - non constant N and L

This implementation is when either N or L are not constant (either a bump-map or illumination from an Omni light or a Spotlight). This implies that R•V must be supplied, and hence $R \bullet V^n$ must also be calculated.

The specular calculation process can be represented as a process with the inputs and outputs as shown in Fig 102. Fig. 103 shows an actual process for calculating the specular contribution where the following constants are set by software:

| Constant | Value |
|---|---|
| $K_1$ | $k_s$ |
| $K_2$ | $k_{sc}$ |
| $K_3$ | $(1-k_{sc})\,I_p$ |

The following lookup table is used:

| Lookup | Size | Details |
|---|---|---|
| $LU_1$ | 32 entries 16 bits per entry | $X^n$<br>Table indexed by 5 highest bits of integer R•V<br>Result by linear interpolation of 2 entries using fraction of R•V. Interpolation by 2 Multiplies.<br>The time taken to retrieve the data from the lookup is 2 * 8 bits * 2 entries = 4 cycles. |

When ambient light is the only illumination

If the ambient contribution is the only light source, the process is very straightforward since it is not necessary to add the ambient light to anything with the overall process being as illustrated in Fig. 104. We can divide the image vertically into 2 sections, and process each half simultaneously by duplicating the ambient light logic (thus using a total of 2 Multiply ALUs and 4 Sequential Iterators). The timing is therefore ½ cycle per pixel for ambient light application.

The typical illumination case is a scene lit by one or more lights. In these cases, because ambient light calculation is so cheap, the ambient calculation is included

with the processing of each light source. The first light to be processed should have the correct $I_ak_a$ setting, and subsequent lights should have an $I_ak_a$ value of 0 (to prevent multiple ambient contributions).

If the ambient light is processed as a separate pass (and not the first pass), it is necessary to add the ambient light to the current calculated value (requiring a read and write to the same address). The process overview is shown in Fig.105.

The process uses 3 Image Iterators, 1 Multiply ALU, and takes 1 cycle per pixel on average.

Infinite Light Source

In the case of the infinite light source, we have a constant light source intensity across the image. Thus both L and $f_{att}$ are constant.

No Bump Map

When there is no bump-map, there is a constant normal vector N [0, 0, 1]. The complexity of the illumination is greatly reduced by the constants of N, L, and $f_{att}$. The process of applying a single Directional light with no bump-map is as illustrated in Fig. 105 where the following constant is set by software:

| Constant | Value |
|---|---|
| $K_1$ | $I_p$ |

For a single infinite light source we want to perform the logical operations as shown in Fig.106 where $K_1$ through $K_4$ are constants with the following values:

| Constant | Value |
|---|---|
| $K_1$ | $K_d(NsL) = K_d L_z$ |
| $K_2$ | $k_{sc}$ |
| $K_3$ | $K_s(NsH)^n = K_s H_z^2$ |
| $K_4$ | $I_p$ |

The process can be simplified since $K_2$, $K_3$, and $K_4$ are constants. Since the complexity is essentially in the

calculation of the specular and diffuse contributions (using 3 of the Multiply ALUs), it is possible to safely add an ambient calculation as the 4th Multiply ALU. The first infinite light source being processed can have the true ambient light parameter $I_a k_a$, and all subsequent infinite lights can set $I_a k_a$ to be 0. The ambient light calculation becomes effectively free.

If the infinite light source is the first light being applied, there is no need to include the existing contributions made by other light sources and the situation is as illustrated in Fig.107 where the constants have the following values:

| Constant | Value |
|---|---|
| $K_1$ | $k_d(LsN) = k_d L_z$ |
| $K_4$ | $I_p$ |
| $K_5$ | $(1- k_s(NsH)^n) I_p = (1 - k_s H_z^n) I_p$ |
| $K_6$ | $k_{sc} k_s (NsH)^n I_p = k_{sc} k_s H_z^n I_p$ |
| $K_7$ | $I_a k_a$ |

If the infinite light source is not the first light being applied, the existing contribution made by previously processed lights must be included (the same constants apply) and the situation is as illustrated in Fig.105.

In the first case 2 Sequential Iterators 490, 491 are required, and in the second case, 3 Sequential Iterators 490, 491, 492 (the extra Iterator is required to read the previous light contributions). In both cases, the application of an infinite light source with no bump map takes 1 cycle per pixel, including optional application of the ambient light.

With Bump Map

When there is a bump-map, the normal vector N must be calculated per pixel and applied to the constant light source vector L. $1/||N||$ is also used to calculate R•V, which is required as input to the Calculate Specular 2 process. The following constants are set by software:

| Constant | Value |
|---|---|
| $K_1$ | $X_L$ |
| $K_2$ | $Y_L$ |
| $K_3$ | $Z_L$ |
| $K_4$ | $I_p$ |

Bump-map Sequential Read Iterator 3 is responsible for reading the current line of the bump-map. It provides the input for determining the slope in X. Bump-map Sequential Read Iterators 1 and are responsible for reading the line above and below the current line. They provide the input for determining the slope in Y.

Omni Lights

In the case of the Omni light source, the lighting vector L and attenuation factor $f_{att}$ change for each pixel across an image. Therefore both L and $f_{att}$ must be calculated for each pixel.

No Bump Map

When there is no bump-map, there is a constant normal vector N [0, 0, 1]. Although L must be calculated for each pixel, both N•L and R•V are simplified to $Z_L$. When there is no bump-map, the application of an Omni light can be calculated as shown in Fig. 107 where the following constants are set by software:

| Constant | Value |
|---|---|
| $K_1$ | $X_P$ |
| $K_2$ | $Y_P$ |
| $K_3$ | $I_p$ |

The algorithm optionally includes the contributions from previous light sources, and also includes an ambient light calculation. Ambient light needs only to be included once. For all other light passes, the appropriate constant in the Calculate Ambient process should be set to 0.

The algorithm as shown requires a total of 19 multiply/accumulates, with 4 Multiply ALUs the task of

illuminating a single pixel can be accomplished in a minimum of 5 cycles. The times taken for the lookups are 1 cycle during the calculation of L, and 4 cycles during the specular contribution. The processing time of 5 cycles is therefore the best that can be accomplished. The time taken is increased to 6 cycles in case it is not possible to optimally microcode the ALUs for the function. The speed for applying an Omni light onto an image with no associated bump-map is 6 cycles per pixel.

With Bump-map

When an Omni light is applied to an image with an associated a bump-map, calculation of N, L, N•L and R•V are all necessary. The process of applying an Omni light onto an image with an associated bump-map is as indicated in Fig. 108 where the following constants are set by software:

| Constant | Value |
|---|---|
| $K_1$ | $X_P$ |
| $K_2$ | $Y_P$ |
| $K_3$ | $I_P$ |

The algorithm optionally includes the contributions from previous light sources, and also includes an ambient light calculation. Ambient light needs only to be included once. For all other light passes, the appropriate constant in the Calculate Ambient process should be set to 0.

The algorithm as shown requires a total of 32 multiply/accumulates. With 4 Multiply ALUs the task of illuminating a single pixel can be accomplished in a minimum of 8 cycles. The times taken for the lookups are 1 cycle each during the calculation of both L and N, and 4 cycles for the specular contribution. However the lookup required for N and L are both the same (thus 2 LUs implement the 3 LUs). The processing time of 8 cycles is therefore the best that can be accomplished. The time taken is extended to 9 cycles in case it is not possible to optimally microcode the ALUs for the function. The speed for applying an Omni light

onto an image with an associated bump-map is 9 cycles per pixel.

Spotlights

Spotlights are similar to Omni lights except that the attenuation factor $f_{att}$ is modified by a cone/penumbra factor $f_{cp}$ that effectively focuses the light around a target.

No bump-map

When there is no bump-map, there is a constant normal vector N [0, 0, 1]. Although L must be calculated for each pixel, both N•L and R•V are simplified to $Z_L$. Fig. 109 illustrates the application of a Spotlight to an image where the following constants are set by software:

| Constant | Value |
|---|---|
| $K_1$ | $X_P$ |
| $K_2$ | $Y_P$ |
| $K_3$ | $I_p$ |

The algorithm optionally includes the contributions from previous light sources, and also includes an ambient light calculation. Ambient light needs only to be included once. For all other light passes, the appropriate constant in the Calculate Ambient process should be set to 0.

The algorithm as shown requires a total of 30 multiply/accumulates. With 4 Multiply ALUs the task of illuminating a single pixel can be accomplished in a minimum of 8 cycles. The times taken for the lookups are 1 cycle during the calculation of L, 4 cycles for the specular contribution, and 2 sets of 4 cycle lookups in the cone/penumbra calculation. The processing time of 8 cycles is therefore the best that can be accomplished. The time taken is extended to 9 cycles in case it is not possible to optimally microcode the ALUs for the function. The speed for applying a Spotlight onto an image with no associated bump-map is 9 cycles per pixel.

With bump-map

When a Spotlight is applied to an image with an

associated a bump-map, calculation of N, L, N•L and R•V are all necessary. The process of applying a single Spotlight onto an image with associated bump-map is illustrated in Fig. 110 where the following constants are set by software:

| Constant | Value |
|---|---|
| $K_1$ | $X_P$ |
| $K_2$ | $Y_P$ |
| $K_3$ | $I_p$ |

The algorithm optionally includes the contributions from previous light sources, and also includes an ambient light calculation. Ambient light needs only to be included once. For all other light passes, the appropriate constant in the Calculate Ambient process should be set to 0.

The algorithm as shown requires a total of 41 multiply/accumulates. With 4 Multiply ALUs the task of illuminating a single pixel can be accomplished in a minimum of 11 cycles. The times taken for the lookups are 1 cycle each during the calculation of both L and N, 4 cycles for the specular contribution, and 2 sets of 4 cycle lookups in the cone/penumbra calculation. However the lookup required for N and L are both the same (thus 4 LUs implement the 5 LUs). The processing time of 11 cycles is therefore the best that can be accomplished. The time taken is extended to 12 cycles in case it is not possible to optimally microcode the ALUs for the function. The speed for applying a Spotlight onto an image with associated bump-map is 12 cycles per pixel.

Serial Interfaces 52 (Fig. 3)- USB serial port interface

This is a standard USB serial port, which is connected to the internal chip low speed bus.

Keyboard interface 55

This is a standard low-speed serial port, which is connected to the internal chip low speed bus.

Authentication chip serial interfaces 64

These are 2 standard low-speed serial ports, which are connected to the internal chip low speed bus. The reason for having 2 ports is to connect to both the on-camera Authentication chip, and to the print-roll Authentication chip using separate lines. Only using 1 line may make it possible for a clone print-roll manufacturer to design a chip which, instead of generating an authentication code, tricks the camera into using the code generated by the authentication chip in the camera.

Parallel Interface 65

The parallel interface connects the ACP to individual static electrical signals. The following is a table of connections to the parallel interface:

| Connection | Direction | Pins |
|---|---|---|
| Paper transport stepper motor | Output | 4 |
| Artcard stepper motor | Output | 4 |
| Zoom stepper motor | Output | 4 |
| Guillotine solenoid | Output | 1 |
| Flash trigger | Output | 1 |
| Status LCD segment drivers | Output | 7 |
| Status LCD common drivers | Output | 4 |
| Artcard illumination LED | Output | 1 |
| Artcard status LED (red/green) | Input | 2 |
| Artcard sensor | Input | 1 |
| Paper pull sensor | Input | 1 |
| Orientation sensor | Input | 2 |
| Buttons | Input | 4 |
| Total | | 36 |

Print Head Interface 62

Once an image has been processed, it can be printed. The Print Head Interface connects the ACP to the Print Head, providing both data and appropriate signals to the external

Print Head.

Print Head 44

Fig. 111 illustrates the logical layout of a single print Head which logically consists of 8 segments, each printing bi-level cyan, magenta, and yellow onto a portion of the page.

Loading a segment for printing

Before anything can be printed, each of the 8 segments in the Print Head must be loaded with 6 rows of data corresponding to the following relative rows in the final output image:

Row 0 = Line N, Yellow, even dots 0, 2, 4, 6, 8, ...
Row 1 = Line N+8, Yellow, odd dots 1, 3, 5, 7, ...
Row 2 = Line N+10, Magenta, even dots 0, 2, 4, 6, 8, ...
Row 3 = Line N+18, Magenta, odd dots 1, 3, 5, 7, ...
Row 4 = Line N+20, Cyan, even dots 0, 2, 4, 6, 8, ...
Row 5 = Line N+28, Cyan, odd dots 1, 3, 5, 7, ...

Each of the segments prints dots over different parts of the page. Each segment prints 750 dots of one color, 375 even dots on one row, and 375 odd dots on another. The 8 segments have dots corresponding to positions:

| Segment | First dot | Last dot |
|---|---|---|
| 0 | 0 | 749 |
| 1 | 750 | 1499 |
| 2 | 1500 | 2249 |
| 3 | 2250 | 2999 |
| 4 | 3000 | 3749 |
| 5 | 3750 | 4499 |
| 6 | 4500 | 5249 |
| 7 | 5250 | 5999 |

Each dot is represented in the Print Head segment by a single bit. The data must be loaded 1 bit at a time by placing the data on the segment's BitValue pin, and clocked in to a shift register in the segment according to BitClock. Since the data is loaded into a shift register, the order of

loading bits must be correct. Data can be clocked in to the Print Head at a maximum rate of 10 MHz.

Once all the bits have been loaded, they must be transferred in parallel to the Print Head output buffer, ready for printing. The transfer is accomplished by a single pulse on the segment's ParallelXferClock pin.

Controlling the Print

In order to conserve power, not all the dots of the Print Head have to be printed simultaneously. A set of control lines enables the printing of specific dots. An external controller, such as the ACP, can change the number of dots printed at once, as well as the duration of the print pulse in accordance with speed and/or power requirements.

Each segment has 5 NozzleSelect lines, which are decoded to select 32 sets of nozzles per row. Since each row has 375 nozzles, each set contains 12 nozzles. There are also 2 BankEnable lines, one for each of the odd and even rows of color. Finally, each segment has 3 ColorEnable lines, one for each of C, M, and Y colors. A pulse on one of the ColorEnable lines causes the specified nozzles of the color's specified rows to be printed. A pulse is typically about 2μs in duration.

If all the segments are controlled by the same set of NozzleSelect, BankEnable and ColorEnable lines (wired externally to the print head), the following is true:
If both odd and even banks print simultaneously (both BankEnable bits are set), 24 nozzles fire simultaneously per segment, 192 nozzles in all, consuming 5.7 Watts.

If odd and even banks print independently, only 12 nozzles fire simultaneously per segment, 96 in all, consuming 2.85 Watts.

Print Head Interface 62

The Print Head Interface 62 connects the ACP to the Print Head, providing both data and appropriate signals to the external Print Head. The Print Head Interface 62 works

in conjunction with both a VLIW processor 74 and a software algorithm running on the CPU in order to print a photo in approximately 2 seconds.

An overview of the inputs and outputs to the Print Head Interface is shown in Fig. 112. The Address and Data Buses are used by the CPU to address the various registers in the Print Head Interface. A single BitClock output line connects to all 8 segments on the print head. The 8 DataBits lines lead one to each segment, and are clocked in to the 8 segments on the print head simultaneously (on a BitClock pulse). For example, dot 0 is transferred to $segment_0$, dot 750 is transferred to $segment_1$, dot 1500 to $segment_2$ etc simultaneously.

The VLIW Output FIFO contains the dithered bi-level C, M, and Y 6000 x 9000 resolution print image in the correct order for output to the 8 DataBits. The ParallelXferClock is connected to each of the 8 segments on the print head, so that on a single pulse, all segments transfer their bits at the same time. Finally, the NozzleSelect, BankEnable and ColorEnable lines are connected to each of the 8 segments, allowing the Print Head Interface to control the duration of the C, M, and Y drop pulses as well as how many drops are printed with each pulse. Registers in the Print Head Interface allow the specification of pulse durations between 0 and 6 µs, with a typical duration of 2µs.

Printing an Image

There are 2 phases that must occur before an image is in the hand of the Artcam user:

1. Preparation of the image to be printed
2. Printing the prepared image

Preparation of an image only needs to be performed once. Printing the image can be performed as many times as desired.

Prepare the Image

Preparing an image for printing involves:

1. Convert the Photo Image into a Print Image
2. Rotation of the Print Image (internal color space) to align the output for the orientation of the printer
3. Up-interpolation of compressed channels (if necessary)
4. Color conversion from the internal color space to the CMY color space appropriate to the specific printer and ink

At the end of image preparation, a 4.5MB correctly oriented 1000 x 1500 CMY image is ready to be printed.

Convert Photo Image to Print Image

The conversion of a Photo Image into a Print Image requires the execution of a Vark script to perform image processing. The script is either a default image enhancement script or a Vark script taken from the currently inserted Artcard. The Vark script is executed via the CPU, accelerated by functions performed by the VLIW Vector Processor.

Rotate the Print Image

The image in memory is originally oriented to be top upwards. This allows for straightforward Vark processing. Before the image is printed, it must be aligned with the print roll's orientation. The re-alignment only needs to be done once. Subsequent Prints of a Print Image will already have been rotated appropriately.

The transformation to be applied is simply the inverse of that applied during capture from the CCD when the user pressed the "Image Capture" button on the Artcam. If the original rotation was 0, then no transformation needs to take place. If the original rotation was +90 degrees, then the rotation before printing needs to be -90 degrees (same as 270 degrees). The method used to apply the rotation is the Vark accelerated Affine Transform function. The Affine Transform engine can be called to rotate each color channel independently. Note that the color channels cannot be rotated in place. Instead, they can make use of the space previously used for the expanded single channel (1.5MB).

Fig. 113 shows an example of rotation of a Lab image where the a and b channels are compressed 4:1. The L channel is rotated into the space no longer required (the single channel area), then the a channel can be rotated into the space left vacant by L, and finally the b channel can be rotated. The total time to rotate the 3 channels is 0.09 seconds. It is an acceptable period of time to elapse before the first print image. Subsequent prints do not incur this overhead.

Up Interpolate and color convert

The Lab image must be converted to CMY before printing. Different processing occurs depending on whether the a and b channels of the Lab image is compressed. If the Lab image is compressed, the a and b channels must be decompressed before the color conversion occurs. If the Lab image is not compressed, the color conversion is the only necessary step. The Lab image must be up interpolated (if the a and b channels are compressed) and converted into a CMY image. A single VLIW process combining scale and color transform

The method used to perform the color conversion is the Vark accelerated Color Convert function. The Affine Transform engine can be called to rotate each color channel independently. The color channels cannot be rotated in place. Instead, they can make use of the space previously used for the expanded single channel (1.5MB).

Print the Image

Printing an image is concerned with taking a correctly oriented 1000 x 1500 CMY image, and generating data and signals to be sent to the external Print Head. The process involves the CPU working in conjunction with a VLIW process and the Print Head Interface.

The resolution of the image in the Artcam is 1000 x 1500. The printed image has a resolution of 6000 x 9000 dots, which makes for a very straightforward relationship: 1 pixel = 6 x 6 = 36 dots. Since each dot is 16.6µm, the 6 x 6 dot square is 100 µm square. Since each of the dots is bi-

level, the output must be dithered.

The image should be printed in approximately 2 seconds. For 9000 rows of dots this implies a time of 222 μs time between printing each row. The Print Head Interface must generate the 6000 dots in this time, an average of 37ns per dot. However, each dot comprises 3 colors, so the Print Head Interface must generate each color component in approximately 12ns, or 1 clock cycle of the ACP (10ns at 100 MHz). One VLIW process is responsible for calculating the next line of 6000 dots to be printed. The odd and even C, M, and Y dots are generated by dithering input from 6 different 1000 x 1500 CMY image lines. The second VLIW process is responsible for taking the previously calculated line of 6000 dots, and correctly generating the 8 bits of data for the 8 segments to be transferred by the Print Head Interface to the Print Head in a single transfer. A CPU process updates registers in the first VLIW process 1 line at a time, an 2 different VLIW processes in order to

Generate C, M, and Y Dots

The input to this process is a 1000 x 1500 CMY image correctly oriented for printing. The image is not compressed in any way. As illustrated in Fig. 115, a VLIW microcode program takes the CMY image, and generates the C, M, and Y pixels required by the Print Head Interface to be dithered.

Generate Merged 8 bit Dot Output

This process, as illustrated in Fig. 116, takes a single line of dithered dots and generates the 8 bit data stream for output to the Print Head Interface via the VLIW Output FIFO. The process requires the entire line to have been prepared, since it requires semi-random access to most of the dithered line at once.

Data Card Reader

Fig. 117, there is illustrated on form of card reader 500 which allows for the insertion of Artcards 9 for reading. Fig. 118 shows an exploded perspective of the reader of Fig. 117. Cardreader 500 is interconnected to a

computer system and includes a CCD reading mechanism located under a covering bar 5. The cardreader 500 includes pinch rollers 506, 507 for pinching an inserted Artcard 9. One of the roller e.g 506 is driven by an Artcard motor 37 for the advancement of the card 9 between the two rollers 506 and 507 at a uniformed speed. The Artcard 9 is passed over a series of LED lights 512 which are encased within a clear plastic mould 514 having a semi circular cross section. The cross section focuses the light from the LEDs eg 512 onto the surface of the card as it passes by the LEDs 512. From the surface it is reflected to a high resolution linear CCD 34 which is constructed to a resolution of approximately 480 dpi. The surface of the Artcard 9 is encoded to the level of approximately 1600 dpi hence, the near CCD 16 supersamples the Artcard surface with an approximately three times multiplier. The Artcard 9 is further driven at a speed such that the linear CCD 516 is able to supersample in the direction of Artcard movement at a rate of approximately 4800 readings per inch. The scanned Artcard CCD data is forwarded from the Artcardreader 500 to ACP 31 for processing. A sensor 49, which can comprise a light sensor acts to detect of the presence of the card 13.

The CCD reader includes a bottom substrate 516, a top substrate 514 which comprises a transparent molded plastic. In between the two substrates is inserted the linear CCD array 34 which comprises a thin long linear CCD array constructed by means of semi-conductor manufacturing processes.

Turning to Fig. 119, there is illustrated a side perspective view, partly in section, of the CCD reader unit. The series of LEDs eg. 512 are operated to emit light when a card 9 is passing across the surface of the CCD reader 34. The emitted light is transmitted through a portion of the top substrate 523. The substrate includes a portion eg. 529 having a curved circumference so as to focus light emitted from LED 512 to a point eg. 532 on the surface of the card

9. The focussed light is reflected from the point 532 towards the CCD array 34. A series of microlenses eg. 534, shown in exaggerated form, are formed on the surface of the top substrate 523. The microlenses 523 act to focus light received across the surface to the focussed down to a point 536 which corresponds to point on the surface of the CCD reader 34 for sensing of light falling on the light sensing portion of the CCD array 34.

A number of refinements of the above arrangement are possible. For example, the sensing devices on the linear CCD 34 may be staggered. The corresponding microlenses 34 can also be correspondingly formed as to focus light into a staggered series of spots so as to correspond to the staggered CCD sensors.

One the data surface area of the Artcard 9 is modulated with a checkerboard pattern as previously discussed with reference to Fig. 30. Other forms of high frequency modulation may be possible however.

It will be evident that an Artcard printer can be provided as for the printing out of data on storage Artcard. Hence, the Artcard system can be utilized as a general form of information distribution outside of the Artcam device. An Artcard printer can prints out Artcards on high quality print surfaces and multiple Artcards can be printed on same sheets and later separated. On a second surface of the Artcard 9 can be printed information relating to the files etc. stored on the Artcard 9 for subsequent storage.

Hence, the Artcard system allows for a simplified form of storage which is suitable for use in place of other forms of storage such as CD ROMs, magnetic disks etc. The Artcards 9 can also be mass produced and thereby produced in a substantially inexpensive for redistribution.

Turning to Fig. 120, there is illustrated the print roll 42 and printhead portions of the Artcam. The paper/film 611 is fed in a continuous "web-like" process

to a printing mechanism 15 which includes further pinch rollers 616 - 619 and a print head 44

The pinch roller 13 is connected to a drive mechanism (not shown) and upon rotation of the print roller 613, paper 611 is forced through the printing mechanism 615 and out of the picture output slot 6. A rotary guillotine mechanism (not shown) is utilised to cut the roll of paper 611 at required photo sizes.

It is therefore evident that the printer roll 42 is responsible for supplying paper 611 to the print mechanism 615 for printing of photographically imaged pictures.

In Fig. 121, there is shown an exploded perspective of the print roll 42. The printer roll 10 includes output printer paper 611 which is output under the operation of pinching rollers 612, 613.

Referring now to Fig. 122, there is illustrated a more fully exploded perspective view, of the print roll 42 of Fig. 121 without the "paper" film roll. The print roll 42 includes three main parts comprising ink reservoir section 620, paper roll sections 622, 623 and outer casing sections 626, 627.

Turning first to the ink reservoir section 620, which includes the ink reservoir or ink supply sections 633. The ink for printing is contained within three bladder type containers 630 - 632. The printer roll 42 is assumed to provide a full colour output inks. Hence, a first ink reservoir or bladder container 630 contains cyan coloured ink. A second reservoir 631 contains magenta coloured ink and a third reservoir 632 contains yellow ink. Each of the reservoirs 630 - 632, although having different volumetric dimensions, are designed to have substantially the same volumetric size.

The ink reservoir sections 621, 633, in addition to cover 624 can be made of plastic sections and are designed to be mated together by means of heat sealing, ultra violet radiation, etc. Each of the equally sized ink

reservoirs 630 - 632 is connected to a corresponding ink channel 639 - 641 for allowing the flow of ink from the reservoir 630 - 632 to a corresponding ink output port 35 - 37. The ink reservoir 632 having ink channel 641, and output port 37, the ink reservoir 31 having ink channel 640 and output port 636, and the ink reservoir 30 having ink channel 639 and output port 637.

In operation, the ink reservoirs 630 - 632 can be filled with corresponding ink and the section 633 joined to the section 621. The ink reservoir sections 630 - 632, being collapsible bladders, allow for ink to traverse ink channels 639 - 641 and therefore be in fluid communication with the ink output ports 635 - 637. Further, if required an air inlet port can also be provided to allow the pressure associated with ink channel reservoirs 630 - 632 to be maintained as required.

The cap 624 can be joined to the ink reservoir section 620 so as to form a pressurised cavity, accessible by the air pressure inlet port.

The ink reservoir sections 621, 633 and 624 are designed to be connected together as an integral unit and to be inserted inside printer roll sections 622, 623. The printer roll sections 622, 623 are designed to mate together by means of a snap fit by means of male portions 645 - 647 mating with corresponding female portions (not shown). Similarly, female portions 654 - 656 are designed to mate with corresponding male portions 660 - 662. The paper roll sections 622, 623 are therefore designed to be snapped together. One end of the film within the role is print role is pinched between the two sections 622, 623 when they are joined together. The print can then be rolled on the print roll sections 622, 625 as required.

As noted previously, the ink reservoir sections 620, 621, 633, 624 are designed to be inserted inside the paper roll sections 622, 623. The printer roll sections 622, 623 are able to be rotatable around stationery ink

reservoir sections 621, 633 and 624 to dispense film on demand.

The outer casing sections 626 and 627 are further designed to be coupled around the print roller sections 622, 623. In addition to each end of pinch rollers eg 612, 613 is designed to clip in to a corresponding cavity eg 670 in cover 626, 627 with roller 613 being driven externally (not shown) to feed the print film and out of the print roll.

Finally, a cavity 677 can be provided in the ink reservoir sections 620, 621 for the insertion and gluing of an silicon chip integrated circuit type device 79 for the storage of information associated with the print roll 42.

As shown in Fig. 121, the print roll 42 is designed to be inserted into the Artcam camera device so as to couple with a coupling unit 680 which includes connecter pads 681 for providing a connection with the silicon chip 53. Further, the connecter 680 includes end connecters of four connecting with ink supply ports 635 - 637. The ink supply ports are in turn to connect to ink supply lines eg 682 which are in turn interconnected to printheads supply ports eg. 687 for the flow of ink to printhead 44 in accordance with requirements.

The "media" 611 utilised to form the roll can comprise many different materials on which it is designed to print suitable images. For example, opaque rollable plastic material may be utilized, transparencies may be used by using transparent plastic sheets, metallic printing can take place via utilisation of a metallic sheet film. Further, fabrics could be utilised within the printer roll 42 for printing images on fabric, although care must be taken that only fabrics having a suitable stiffness or suitable backing material are utilised.

When the print media is plastic, it can be coated with a layer which fixes and absorbs the ink. Further,

several types of print media may be used, for example, opaque white matte, opaque white gloss, transparent film, frosted transparent film, lenticular array film for stereoscopic 3D prints, metallised film, film with the embossed optical variable devices such as gratings or holograms, media which is pre-printed on the reverse side, and media which includes a magnetic recording layer. When utilising a metallic foil, the metallic foil can have a polymer base, coated with a thin (several micron) evaporated layer of aluminium or other metal and then coated with a clear protective layer adapted to receive the ink via the ink printer mechanism.

In use the print roll 42 is obviously designed to be inserted inside a camera device so as to provide ink and paper for the printing of images on demand. The ink output ports 635 - 637 meet with corresponding ports within the camera device and the pinch rollers 672, 673 are operated to allow the supply of paper to the camera device under the control of the camera device.

As illustrated in Fig. 122, a mounted silicon chip 53 is insert in one end of the print roll 42. In Fig. 123 the authentication chip 53 is shown in more detail and includes four communications leads 680 - 683 for communicating details from the chip 53 to the corresponding camera to which it is inserted.

Turning to Fig. 123, the chip can be separately created 79 by means of encasing a small integrated circuit 687 in epoxy and running bonding leads eg. 688 to the external communications leads 680 - 683. The integrated chip 87 being approximately 400 microns square with a 100 micron scribe boundary. Subsequently, the chip 53 can be glued to an appropriate surface of the cavity of the print roll 42. In Fig. 124, there is illustrated the integrated circuit 87 interconnected to bonding pads 81, 82 in an exploded view of the arrangement of Fig..123.

Referring now to Fig. 125, there is illustrated

generally 700 the internal architecture of the chip 53 of Fig. 123 The chip architecture 700 includes a flash memory store 701, a roll authentication unit 702, a command decoder 703 and a communications controller 704.

The communications controller 704 is interconnected to the serial input and output wires 681, 682 for communication with the Artcam. The command coder 703 receives commands from the camera 30 via the communications controller 704 controls the flash memory store 701 and roll authentication unit 702 to carry out the command. Preferably, the flash memory store 701 provides 1,024 bits of information which includes fixed data written into the flash memory at manufacturing time in addition to variable data storage.

Turning now to Fig. 126, there is illustrated 705 the information stored within the flash memory store 701. This data can include the following:

Factory Code

The factory code is a 16 bit code indicating the factory at which the print roll was manufactured. This identifies factories belonging to the owner of the print roll technology, or factories making print rolls under license. The purpose of this number is to allow the tracking of factory that a print roll came from, in case there are quality problems.

Batch Number

The batch number is a 32 bit number indicating the manufacturing batch of the print roll. The purpose of this number is to track the batch that a print roll came from, in case there are quality problems.

Serial Number

A 48 bit serial number is provided to allow unique identification of each print roll up to a maximum of 280 trillion print rolls.

Manufacturing date

A 16 bit manufacturing date is included for tracking the age of print rolls, in case the shelf life is limited.

Media length

The length of print media remaining on the roll is represented by this number. This length is represented in small units such as millimetres or the smallest dot pitch of printer devices using the print roll and to allow the calculation of the number of remaining photos in each of the well known C, H, and P formats, as well as other formats which may be printed. the use of small units also ensures a high resolution can be used to maintain synchronisation with pre-printed media.

Media Type

The media type datum enumerates the media contained in the print roll.

(1) Transparent
(2) Opaque white
(3) Opaque tinted
(4) 3D lenticular
(5) Pre-printed: length specific
(6) Pre-printed: not length specific
(7) Metallic foil
(8) Holographic/optically variable device foil

Pre-printed Media Length

The length of the repeat pattern of any pre-printed media contained, for example on the back surface of the print roll is stored here.

Ink Viscosity

The viscosity of each ink colour is included as an 8 bit number. the ink viscosity numbers can be used to adjust the print head actuator characteristics to compensate for viscosity (typically, a higher viscosity will require a longer actuator pulse to achieve the same drop volume).

Recommended Drop Volume for 1200 dpi

The recommended drop volume of each ink colour is included as an 8 bit number. The most appropriate drop volume will be dependant upon the ink and print media characteristics. For example, the required drop volume will

decrease with increasing dye concentration or absorptivity. Also, transparent media require around twice the drop volume as opaque white media, as light only passes through the dye layer once for transparent media.

As the print roll contains both ink and media, a custom match can be obtained. The drop volume is only the recommended drop volume, as the printer may be other than 1200 dpi, or the printer may be adjusted for lighter or darker printing.

Ink Colour

The colour of each of the dye colours is included and can be used to "fine tune" the digital half toning that is applied to any image before printing.

Remaining Media Length Indicator

The length of print media remaining on the roll is represented by this number and is updatable by the camera device. The length is represented in small units (eg. 1200 dpi pixels) to allow calculation of the number of remaining photos in each of C, H, and P formats, as well as other formats which may be printed. The high resolution can also be used to maintain synchronization with pre-printed media.

Copyright or Bit Pattern

This 512 bit pattern represents an ASCI character sequence sufficient to allow the contents of the flash memory store to be copyrightable.

Authentication Key

This key includes authentication data to make it difficult for third parties to reverse engineer the print roll technology.

Finally, a further 88 bits are reserved for future camera use.

The role authentication unit 702 as will become more apparent hereinafter, takes the authentication key from flash memory store 701 and combines it with a print roll test code received from the camera processor.

Authentication

The print roll manufacturing process includes in-built measures to stop illegal clone manufacture in countries with weak industrial property protection from copying the technology.

The print rolls 42 are not protected against cloning by high technology barriers, such as the extraordinarily difficult chemistry of colour silver halide film in photographic reproduction. The present embodiment is simply constructed from plastic injection moulding, coated paper, and ink. the coated paper and ink may only be required to be compatible, and do not need to match some special formulation. To protect against these problems, an authentication code and circuit is included in the print roll chip.

The authentication system prevents illegal copy ing which can have the disastrous consequence of ink nozzles becoming clogged by poorly filtered ink in "clone" print rolls. This will assist in stopping a consumer blaming the camera manufacturer and in stopping the spread of counterfeit print rolls.

The authentication system should remain outside most countries' legislation in respect of the export of cryptographic materials.

(1) Reverse Engineering of the Print Roll Chip

The best way to protect against reverse engineering of the chip is to make the benefit of reverse engineering minimal. To achieve this, the authentication keys are stored in non-volatile flash memory store 101 not in ROM.

(2) Brute force cryptanalysis

Brute force cryptanalysis can be prevented by making the authentication key long enough. To be secure against computational improvements over the next fifty years, a long key is necessary. A key length of 128 bits means that $2^{128}$ tests ($3.4 \times 10^{38}$ tests) must be made to launch a brute force attack. This would take ten billion years on an array or a trillion processors each running 1 billion tests per second.

(3) Substitution with a complete lookup table

If the number of test codes sent by the camera to the print roll is small, then there is no need for a clone manufacturer to crack the authentication code. Instead, the clone manufacturer could incorporate a ROM in their print roll which had a record of all of the responses from a genuine print roll to the codes sent by the camera. In 30 years, it may be cost effective to build a terabyte ROM into each clone print roll. Therefore, the camera should send random authentication test words that are at least 40 bits long. A 128 bit authentication test word will certainly be more than adequate.

(4) Substitution with a sparse lookup table

If the test codes sent by the camera are somehow predictable, rather than effectively random, then the clone manufacturer need not provide a complete lookup table. For example:

(a) If the test code is simply the serial number of the camera, the clone manufacturer need simply provide a lookup table which contains values for past and predicted future serial camera serial numbers. There are unlikely to be more than $10_9$ of these.

(b) If the test code is simply the date, then the clone manufacturer can produce a lookup table using the date as the address.

(c) If the test code is a pseudo-random number using either the serial number or the date as a see, then the clone manufacturer just needs to crack the pseudo-random number generator in the camera. This is probably not difficult, as the clone manufacturer may gain access to the object code of the camera. The clone manufacturer could then produce an content addressable memory (or other sparse array lookup) using these codes to access stored authentication codes.

Therefore, long random test keys should be generated by a relatively secure process. This random number generator

can be in the machine which writes the authentication code to the chips.

(5) Usurping the authentication comparison process

It must be assumed that a clone manufacturer will have access to both the camera and the print roll designs. It should not be possible for the clone manufacturer to design a chip which, instead of generating an authentication code, tricks the camera into using the code generated by the duplicate authentication chip in the camera. This can be achieved by providing separate serial data Tx and Rx lines for the camera and print roll authentication chips.

(6) Differential Cryptanalysis

It is important that the system adopted is secure against differential cryptanalysis. Differential cryptanalysis is a well known technique where pairs of input streams are generated with known differences, and the differences in the encoded streams are analysed. A small amount ($10^6$ or so) of weakening could be accepted.

(7) Listening to the data flow between the camera and the print roll

Again a logic analyser can be connected to the data stream between the camera and the print roll. In this way, the codes sent to the print roll, and the authentication reply, can be monitored. However, these codes are 128 bit pseudo-random numbers, which are only related by the encoding algorithm in the authentication chips. This is essentially a known plaintext attack, which is less powerful than a chosen plaintext attack.

(8) Direct viewing of chip operation

If the chip operation could be directly viewed using an STM or an electron beam, the authentication codes could be recorded as they are read from the internal non-volatile memory and loaded into internal registers on the chip.

(9) Direct viewing of the non-volatile memory

If the chip where sliced to that the floating gates of the Flash memory were exposed, without discharging them,

then the authentication key could probably be viewed directly using an STM. However, slicing the chip to this level without discharging the gates is difficult. Using wet etching, plasma etching, ion milling, or chemical mechanical polishing will almost certainly discharge the small charges present on the floating of the chip gates.

(10) Viewing $I_{dd}$ fluctuations

Whenever the Authentication key is being read from memory, the Message Authentication Code (MAC) circuitry is also operating, obscuring the $I_{dd}$ signal.

Also, after the code words have been programmed, a lock bit is programmed which prevents subsequent programming of the code words. This prevents detection of the code words by monitoring the difference in $I_{dd}$ that may occur when programming over either a high or a low bit.

(11) Bribery and other industrial espionage

It is not necessary for any human to know, or to be able to find out, what the authentication numbers are. Therefore, the numbers are safe from bribery or other corruption.

There need only be one or a few machines which programs the print roll chips, and these machines could be kept secure, preventing their theft. Authentication chips may be stolen en-route to print roll factories, but this would only enable the manufacture of as many clone print rolls as there were chips stolen, which would probably not exceed a few million in any one shipment. It would not be viable for a print roll illegal clone manufacturer to continually steal chips.

(12) Reverse engineering the authentication key generator

If the clone manufacture can obtain the code for the authentication key generator, then this could be reverse engineered. For maximum security, the Authentication key should be truly random. This is simply achieve by flipping a coin 128 times, and entering the key into the authentication chip programmer is a secure environment.

This only has to be done once.

(13) Management decision to omit authentication to save costs

Without any form of protection, illegal cloning is almost certain. However, with the patent and copyright protection, the probability of illegal cloning may be reduced to say 50%.

However, this is not the only loss possible. If a clone manufacturer were to introduce clone print rolls which caused damage to the camera (eg. clogged nozzles), then the loss in market acceptance, and the expense of warranty repairs, may also be significant. Upon insertion of a print roll, the ACP 31 interrogates a print roll chip 53 in addition to interrogating an exact replica of the chip 54 stored within the camera system. The print roll chip 53 is designed to be fed a print roll test code to which it applied a one way hash function to produce a resultant code that is checked by the camera processor 105 which also sends the same code to its camera authorisation chip 106.

Turning now to Fig. 127, there is illustrated the significant steps in the authorisation method of the preferred embodiment. Each Artcam is provided with a unique random identification code 710. The Artcam processor takes the identification code 710 and a current time value 711 from the real time clock of the Artcam processor and exclusive ORs them together 712. The result of this process is utilised as a seed to a random number generator 714 which produces a print roll test code having 128 bits. The Artcam processor then transmits the print roll test code to the Artcam authorisation chip 54 and the print roll authorisation chip 53 which each utilises their internally stored key via a corresponding roll authentication unit 702 (Fig. 125) to return to the Artcam processor 31 at stage 719 the expected output values for the given input value. The Artcam processor checks to values to assure they are the same and accepts or rejects the print roll based on the

quality of the two values.

It will be evident from the forgoing it is crucial that the key utilised by the Artcam authorisation chip 54 and print roll authorisation chip 53 is kept secret. As previously noted, the authorization key is stored in the flash memory store 709 of the print roll authorisation chip. Therefore, an attack by way of reverse engineering of the chip will lead to minimal results. One form of attack may be to monitor the chips operation utilising a scanning tunnelling microscope (STM) or an electron beam to monitor the authorisation codes as they are read from the internal non-volatile memory and loaded into internal registers on the chip. Turning now to Fig. 128, such analysis can be circumvented by incorporated a shielding metal layer 725, over the lower circuitry, as an extra metallisation layer.

Of course, the attacker may simply chose to wet etch the metal layer 725. However, if the metal layer 725 is utilised as the ground plane for connections within the chip circuitry, the metallisation layer, if removed, will result in the chip seeking to malfunction, thereby preventing reverse analysis. This means the attacker is forced to either remove the metal layer and lay new ground connections or to mask the metal layer before removal. Masking of the metal layer for removal is the easiest of these two processes but will still be very difficult. In this case, the attacker must:

(1) reverse engineer the chip to find out where the ground connections should be;

(2) create a mask corresponding to the required ground plane pattern connection;

(3) apply a photo resist to the chip. This will be extremely difficult as the individual chip is only approximately 400 microns square. Therefore, standard semi-conductor processes of applying a photo

resist, in particular resist spin processing, cannot be utilised;

(4) soft bake the resist;

(5) expose the resist. This will again be difficult for a single chip as modern lithographic equipment is designed for a wafer;

(6) hard bake the resist; and

(7) etch the top metallisation layer.

The process of high temperature resist baking will most likely destroy the charge patterns in the non-volatile memory which holds the authentication numbers making this process fruitless.

Further, viewing the date flow in the chip can be made even more difficult by making all the connections from which is possible to view the authentication numbers in the polysilicon layer.

The authentication key should be truly random, to prevent compromise by obtaining knowledge of the process used to generate the authentication key. A simple way is for a trusted human to flip a coin 128 times, while entering 0 (heads) or 1 (tails) into the keyboard in a secure environment. The authentication key need only be known by the machine which programs the authentication chips (the human coin flipper will not remember it). So that this machine cannot be stolen, all authentication numbers and chips should be programmed in one place, and shipped to different print roll and Artcam manufacturing sites. Other data specific to a Artcam or print roll can be programmed at this place of manufacture.

Of course, it is necessary to ensure that the authentication key is never lost, as this would prevent the legitimate manufacture of compatible print rolls. Further, the bit pattern preferably contains clearly copyrightable material such that the attacker in order to replicate the operation of the authorisation chip 53 must also copy the bit pattern and therefore is likely to

infringe copyright laws in various jurisdictions. The bit pattern is preferably the original work of an identifiable author reduced to a tangible form. For example, it could be a particular image of bits, otherwise it could be a corresponding ASCII equivalent of prose. Further, it should represent the application of knowledge, judgement, skill and labour by the author. It should not be an integral part of the chip but merely stored in the chips memory. Of course, preferably, the copyright ownership of the bit pattern resides with the print roll manufacturer. As an example, the bit pattern could be an ASCII representation of a short poem. Hence, the allocation of 512 bits should be sufficient. Although the bit pattern could be stored as ROM on the chips , as these chips already have flash memory, the smallest chip size may be achieved by implementing the bit pattern in the flash memory.

Turning now to Fig. 129, there is illustrated the storage table 730 of the Artcam authorisation chip. The table includes manufacturing code, batch number and serial number and date which have an identical format to that previously described. The table 730 also includes information 731 on the print engine within the Artcam device. The information stored can include a print engine type, the DPI resolution of the printer and a printer count of the number of prints produced by the printer device.

Further, an authentication test key 710 is provided which can randomly vary from chip to chip and is utilised as the Artcam random identification code in the previously describe algorithm. The 128 bit print roll authentication key 713 is also provided and is equivalent to the key stored within the print rolls. Next, the 512 bit pattern is stored followed by a 120 bit spare area suitable for Artcam use.

As noted previously, the Artcam preferably includes a liquid crystal display 15 which indicates the number of prints left on the print roll stored within the Artcam. Further, the Artcam also includes a three state switch 17 which allows a user to switch between three standard formats C H and P (classic, HDTV and panoramic). Upon switching between the three states, the liquid crystal display 15 is updated to reflect the number of images left on the print roll if the particular format selected is used.

In order to correctly operate the liquid crystal display, the Artcam processor, upon the insertion of a print roll and the passing of the authentication test reads the from the flash memory store of the print roll chip 53 and determines the amount of paper left. Next, the value of the output format selection switch 17 is determined by the Artcam processor. Dividing the print length by the corresponding length of the selected output format the Artcam processor determines the number of possible prints and updates tthe liquid crystal display 15 with the number of prints left. Upon a user changing the output format selection switch 17 the Artcam processor 105 re-calculates the number of output pictures in accordance with that format and again updates the LCD display 15.

The storage of process information in the printer roll table 705 also allows the Artcam device to take advantage of changes in process and print characteristics of the print roll.

In particular, the pulse characteristics applied to each nozzle within the print head can be altered to take into account of changes in the process characteristics. Turning now to Fig. 130, the Artcam Processor can be adapted to run a software program stored in an ancillary memory chip. The software program, a pulse profile characteriser 771 is able to read a number of variables from the printer roll. These variables include the

remaining roll media on printer roll 772, the printer media type 773, the ink colour viscosity 774, the ink colour drop volume 775 and the ink colour 776. Each of these variables are read by the pulse profile characteriser and a corresponding, most suitable pulse profile is determined in accordance with prior trial and experiment. The parameters alters the printer pulse received by each printer nozzle so as to improve the stability of ink output.

It will be evident that the authorization chip includes significant advances in that important and valuable information is stored on the printer chip with the print roll. This information can include process characteristics of the print roll in question in addition to information on the type of print roll and the amount of paper left in the print roll. Additionally, the print roll interface chip can provide valuable authentication information and can be constructed in a tamper proof manner. Further, a tamper resistant method of utilising the chip has been provided. The utilisation of the print roll chip also allows a convenient and effective user interface to be provided for an immediate output form of Artcam device able to output multiple photographic formats whilst simultaneously able to provide an indicator of the number of photographs left in the printing device.

Print Head Unit

Turning now to Fig. 131 , there is illustrated an exploded perspective view, partly in section, of the print head unit 615 of Fig. 120.

The print head unit 615 is based around the printhead 44 which ejects ink drops on demand on to print media 611 so as to form an image. The print media 611 is pinched between two set of rollers comprising a first set 618, 616 and second set 617, 619.

The printhead 44 operates under the control of power, ground and signal lines 810 which provides power and control

for the printhead 44 and are bonded by means of Tape Automated Bonding (TAB) to the surface of the print printhead 44..

Importantly, the printhead 44 which can be constructed from a silicon wafer device suitably separated, relies upon a series of anisotropic etches 812 through the wafer having near vertical side walls. The through wafer etches 812 allow for the direct supply of ink to the printhead surface from the back of the wafer for subsequent ejection.

The ink is supplied to the back of the inkjet printhead 44 by means of inkhead supply unit 814. The inkjet printhead 44 has three separate rows along its surface for the supply of separate colours of ink. The inkhead supply unit 814 also includes a lid 815 for the sealing of ink channels.

In Figs. 132 - 135, there is illustrated various perspective views of the inkhead supply unit 814. Each of Figs. 132 - 135 illustrate only a portion of the ink head supply unit which can be constructed of indefinite length, the portions shown so as to provide exemplary details. In Fig. 132, there is illustrated a bottom perspective view, Fig. 133 illustrates a top perspective view, Fig. 134 illustrates a close up bottom perspective view, partly in section, Fig. 135 illustrates a top side perspective view showing details of the ink channels, and Fig. 136 illustrates a top side perspective view as does Fig. 137.

There is considerable cost advantage in forming inkhead supply unit 814 from injection moulded plastic instead of, say, micromachined silicon. The manufacturing cost of a plastic ink channel will be considerably less in volume and manufacturing is substantially easier. The design illustrated in the accompanying drawings assumes a 1600 dpi three color monolithic print head, of a predetermined length. The provided flow rate calculations are for a 100mm photo printer.

The inkhead supply unit 814 contains all of the

required fine details. The lid 815 (Fig. 131) is permanently glued or ultrasonically welded to the inkhead supply unit 814 and provides a seal for the ink channels.

Turning to Fig 132, the cyan, magenta and yellow ink flows in through ink inlets 820-822, the magenta ink flows through the throughholes 824,825 and along the magenta main channels 826,827 (Fig. 133). The cyan ink flows along cyan main channel 830 and the yellow ink flows along the yellow main channel 831. As best seen from Fig. 134, the cyan ink in the cyan main channels then flows into a cyan subchannel 833. The yellow subchannel 834 similarly receiving yellow ink from the yellow main channel 831.

As best seen in Fig. 135, the magenta ink also flows from magenta main channels 826,827 through magenta throughholes 836, 837. Returning again to Fig. 134, the magenta ink flows out of the throughholes 836, 837. The magenta ink flows along first magenta subchannel e.g. 838 and then along second magenta subchannel e.g. 839 before flowing into a magenta trough 840. The magenta ink then flows through magenta vias e.g. 842 which are aligned with corresponding inkjet head throughholes (e.g. 812 of Fig. 131) wherein they subsequently supply ink to inkjet nozzles for printing out.

Similarly, the cyan ink within the cyan subchannel 833 flows into a cyan pit area 849 which supplies ink two cyan vias 843, 844. Similarly, the yellow subchannel 834 supplies yellow pit area 46 which in turn supplies yellow vias 847, 848.

As seen in Fig. 135, the printhead is designed to be received within printhead slot 850 with the various vias e.g. 851 aligned with corresponding through holes eg. 851 in the printhead wafer.

Returning to Fig. 131, care must be taken to provide adequate ink flow to the entire printhead chip 44, while satisfying the constraints of an injection molding process. The size of the ink through wafer holes 812 at the back of

the print head chip is approximately 100µm x 50µm, and the spacing between through holes carrying different colors of ink is approximately 170µm. While features of this size can readily be moulded in plastic (compact discs have micron sized features), ideally the wall height must not exceed a few times the wall thickness so as to maintain adequate stiffness. The preferred embodiment overcomes these problems by using hierarchy of progressively smaller ink channels.

In Fig. 136, there is illustrated a wire frame view of a small portion 870 of the surface of the printhead 44. The surface is divided into 3 series of nozzles comprising the cyan series 871, the magenta series 872 and the yellow series 873. Each series of nozzles is further divided into two rows eg. 875, 876 with the printhead 44 having a series of bond pads 878 for bonding of power and control signals.

The print head is preferably constructed in accordance with a large number of different forms of ink jet invented for uses including Artcam devices. A full list of the different invented ink jet types is as set out in the associated Australian Provisional Patent Applications as set out appendix A attached hereto, the applications being filed concurrently herewith. In particular, the present embodiment assumes the ink jet as set out in associated Australian Provisional Patent Application entitled "Image Creation Method and Apparatus (IJ30)" has been utilised.

The printhead nozzles include the ink supply channels 880, equivalent to anisotropic etch hole 812 of Fig. 131. The ink flows from the back of the wafer through supply channel 881 and in turn through the filter grill 882 to ink nozzle chambers eg. 883. The operation of the nozzle chamber 883 and printhead 44 (Fig. 1) is, as mentioned previously, described in the abovementioned patent specification.

Ink Channel Fluid Flow Analysis

Turning now to an analysis of the ink flow, the main

ink channels 826, 827, 830, 831 (Fig.132, Fig. 133) are around 1mm x 1mm, and supply all of the nozzles of one color. The subchannels 833, 834, 838, 839 (Fig. 134) are around 200μm x 100μm and supply about 25 inkjet nozzles each. The print head through holes 843, 844, 847, 848 and wafer through holes eg. 881 (Fig. 136) are 100μm x 50μm and, supply 3 nozzles at each side of the print head through holes. Each nozzle filter 882 has 8 slits, each with an area of 20μm x 2μm and supplies a single nozzle.

An analysis has been conducted of the pressure requirements of an ink jet printer constructed as described. The analysis is for a 1,600 dpi three color process print head for photograph printing. The print width was 100 mm which gives 6,250 nozzles for each color, giving a total of 18,750 nozzles.

The maximum ink flow rate required in various channels for full black printing is important. It determines the pressure drop along the ink channels, and therefore whether the print head will stay filled by the surface tension forces alone, or, if not, the ink pressure that is required to keep the print head full.

To calculate the pressure drop, a drop volume of 2.5 pl for 1,600 dpi operation was utilized . While the nozzles may be capable of operating at a higher rate, the chosen drop repetition rate is 5 KHz which is suitable to print a 150 mm long photograph in an little under 2 seconds. Thus, the print head, in the extreme case, has a 18,750 nozzles, all printing a maximum of 5,000 drops per second. This ink flow is distributed over the hierarchy of ink channels. Each ink channel effectively supplies a fixed number of nozzles when all nozzles are printing.

The pressure drop $\Delta p$ was calculated according to the Darcy-Weisbach formula:

$$\Delta p = \frac{\rho U^2 \, fL}{2D}$$

Where ρ is the density of the ink, $U$ is the average flow velocity, $L$ is the length, $D$ is the hydraulic diameter, and $f$ is a dimensionless friction factor calculated as follows:

$$f = \frac{k}{\mathrm{Re}}$$

Where Re is the Reynolds number and $k$ is a dimensionless friction coefficient dependant upon the cross section of the channel calculated as follows:

$$\mathrm{Re} = \frac{UD}{\nu}$$

Where ν is the kinematic viscosity of the ink.

For a rectangular cross section, $k$ can be approximated by:

$$k = \frac{64}{\frac{2}{3} + \frac{11b}{24a} \quad \frac{11b}{24a}\,(2 - b/a)}$$

Where $a$ is the longest side of the rectangular cross section, and $b$ is the shortest side. The hydraulic diameter $D$ for a rectangular cross section is given by:

$$D = \frac{2ab}{a + b}$$

Ink is drawn off the main ink channels at 250 points along the length of the channels. The ink velocity falls linearly from the start of the channel to zero at the end of the channel, so the average flow velocity $U$ is half of the maximum flow velocity. Therefore, the pressure drop along the main ink channels is half of that calculated using the maximum flow velocity

Utilizing these formulas, the pressure drops can be calculated in accordance with the following tables:

Table of Ink Channel Dimensions and Pressure Drops

| | Number of Items | Length | Width | Depth | Nozzles supplied | Max. ink flow at 5KHz (U) | Pressure drop Δρ |
|---|---|---|---|---|---|---|---|
| Central Molding | 1 | 106mm | 6.4mm | 1.4mm | 18,750 | 0,23ml/seC | NA |
| Cyan main channel (830) | 1 | 100mm | 1mm | 1mm | 6,250 | 0.16µl/µs | 111 Pa |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Magenta main channel (826) | 2 | 100mm | 700μm | 700μm | 3,125 | 0.16μl/μs | 231 Pa |
| Yellow main channel (831) | 1 | 100mm | 1mm | 1mm | 6,250 | 0.16μl/μs | 111 Pa |
| Cyan sub-channel (833) | 250 | 1.5mm | 200μm | 100μm | 25 | 0.16μl/μs | 41.7 Pa |
| Magenta sub-channel (834)(a) | 500 | 200μm | 50μm | 100μm | 12.5 | 0,031μl/μs | 44.5 Pa |
| Magenta sub-channel (838)(b) | 500 | 400μm | 100μm | 200μm | 12.5 | 0.031μl/μs | 5.6 Pa |
| Yellow sub-channel (834). | 250 | 1.5mm | 200μm | 100μm | 25 | 0.016μl/μs | 41.7 Pa |
| Cyan pit (842) | 250 | 200μm | 100μm | 300μm | 25 | 0.010μl/μs | 3.2 Pa |
| Magenta through (840) | 500 | 200μm | 50μm | 200μm | 12.5 | 0.016μl/μs | 18.0 Pa |
| Yellow pit (846) | 250 | 200μm | 100μm | 300μm | 25 | 0.010μl/μs | 3.2 Pa |
| Cyan via (843) | 500 | 100μm | 50μm | 100μm | 12.5 | 0.031μl/μs | 22.3 Pa |
| Magenta via (842) | 500 | 100μm | 50μm | 100μm | 12.5 | 0.031μl/μs | 22.3 Pa |
| Yellow via | 500 | 100μm | 50μm | 100μm | 12.5 | 0.031μl/μs | 22.3 Pa |
| Magenta through hole (837) | 500 | 200μm | 500μm | 100μm | 12.5 | 0.0031μl/μs | 0.87 Pa |
| Chip slot | 1 | 100mm | 730μm | 625 | 18,750 | NA | NA |
| Print head through holes (881)(in the chip substrate) | 1500 | 600μ | 100μm | 50μm | 12.5 | 0.052μl/μs | 133 Pa |
| Print head channel segments (on chip front) | 1,000/ color | 50μm | 60μm | 20μm | 3.125 | 0.049μl/μs | 62.8 Pa |
| Filter Slits (on entrance to nozzle chamber (882) | 8 per nozzle | 2μm | 2μm | 20μm | 0.125 | 0.039μl/μs | 251 Pa |
| Nozzle chamber (on chip front)(883) | 1 per nozzle | 70μm | 30μm | 20μm | 1 | 0.021μl/μs | 75.4 Pa |

The total pressure drop from the ink inlet to the nozzle is therefore approximately 701Pa for cyan and yellow, and 845 Pa for magenta. This is less than 1% of atmospheric

pressure. Of course, when the image printed is less than full black, the ink flow (and therefore the pressure drop) is reduced from these values.

Making the Mold for the Inkhead Supply Unit

The ink head supply unit 14 (Fig. 1) has features as small as 50μ and a length of 106mm. It is impractical to machine the injection molding tools in the conventional manner. However, even though the overall shape may be complex, there are no complex curves required. The injection molding tools can be made using conventional milling for the main ink channels and other millimetre scale features, with a lithographically fabricated inset for the fine features. A LIGA process can be used for the inset.

A single injection molding tool could readily have 50 or more cavities. Most of the tool complexity is in the inset.

Turning to Fig. 131, the printing system is constructed via molding ink supply unit 814 and lid 815 together and sealing them together as previously described. Subsequently printhead 44 is placed in its corresponding slot 850. Adhesive sealing strips 852, 853 are placed over the magenta main channels so to ensure they are properly sealed. The Tape Automated Bonding (TAB) strip 810 is then connected to the inkjet printhead 44 with the tab bonding wires running in the cavity 855. As can best be seen from Fig 136 and 1377, aperture slots are 855 - 862 are provided for the snap in insertion of rollers. The slots provided for the "clipping in" of the rollers with a small degree of play subsequently being provided for simple rotation of the rollers.

In Figs. 138 - 142, there are illustrated various perspective views of the internal portions of a finally assembled Artcam device with devices appropriately numbered.

- Fig. 138 illustrates a top side perspective view of the internal portions of an Artcam camera, showing the parts flattened out;

- Fig. 139 illustrates a bottom side perspective view of the internal portions of an Artcam camera, showing the parts flattened out;
- Fig. 140 illustrates a first top side perspective view of the internal portions of an Artcam camera, showing the parts as encased in an Artcam;
- Fig. 141 illustrates a second top side perspective view of the internal portions of an Artcam camera, showing the parts as encased in an Artcam;
- Fig. 142 illustrates a second top side perspective view of the internal portions of an Artcam camera, showing the parts as encased in an Artcam;

Postcard Print Rolls

Turning now to Fig. 151, in the preferred embodiment, the output printer paper 11 can, on the side that is not to receive the printed image, contain a number of pre-printed "postcard" formatted backing portions 885. The postcard formatted sections 885 can include prepaid postage "stamps" 886 which can comprise a printed authorisation from the relevant postage authority within whose jurisdiction the print roll is to be sold or utilised. By agreement with the relevant jurisdictional postal authority, the print rolls can be made available having different postages. This is especially convenient where overseas travellers are in a local jurisdiction and wishing to send a number of postcards to their home country. Further, an address format portion 87 is provided for the writing of address dispatch details in the usual form of a postcard. Finally, a message area 887 is provided for the writing of a personalised information.

Turning now to Fig. 151 and Fig. 151, the operation of the camera device is such that when a series of images 890-892 is printed on a first surface of the print roll, the corresponding backing surface is that illustrated in Fig. 153. Hence, as each image eg. 890 is printed by the camera,

the back of the image has a ready made postcard 885 which can be immediately despatched at the nearest post office box within the jurisdiction. In this way, personalised postcards can be created.

It would be evident that when utilising the postcard system as illustrated in Fig. 151 and Fig. 152 only predetermined image sizes are possible as the synchronisation between the backing postcard portion 885 and the front image 891 must be maintained. This can be achieved by utilising the memory portions of the authentication chip stored within the print roll to store details of the length of each postcard backing format sheet 885. This can be achieved by either having each postcard the same size or by storing each size within the print rolls on-board print chip memory.

The Artcam camera control system can ensure that, when utilising a print roll having pre-formatted postcards, that the printer roll is utilised only to print images such that each image will be on a postcard boundary. Of course, a degree of "play" can be provided by providing boarder regions at the edges of each photograph which can account for slight misalignment.

Turning now to Fig. 153, it will be evident that postcard rolls can be pre-purchased by a camera user when travelling within a particular jurisdiction where they are available. The postcard roll can, on its external surface, have printed information including country of purchase, the amount of postage on each postcard, the format of each postcard (for example being C,H or P or a combination of these image modes), the countries that it is suitable for use with and the postage expiry date after which the postage is no longer guaranteed to be sufficient can also be provided.

Hence, a user of the camera device can produce a postcard for dispatch in the mail by utilising their hand held camera to point at a relevant scene and taking a

picture having the image on one surface and the pre-paid postcard details on the other. Subsequently, the postcard can be addressed and a short message written on the postcard before its immediate dispatch in the mail.

It would be appreciated by a person skilled in the art that numerous variations and/or modifications may be made to the present invention as shown in the specific embodiment without departing from the spirit or scope of the invention as broadly described. The present embodiment is, therefore, to be considered in all respects to be illustrative and not restrictive.

The present provisional is one of a series of Australian Provisional Patent Applications which relate to a new form of technology for the production of images. These Australian Provisional Patent Applications encompass a broad range of fields and as such, the present provisional is best viewed in the overall context of the development of this new form of technology. Appendix A attached hereto sets out the details of each of the series of Australian Provisional Patent Applications and, to the extent necessary, the associated Australian Provisional Patent Applications are hereby incorporated by cross-reference.

We claim:

1. A camera system comprising:

at least one area image sensor for imaging a scene;

a camera processor means for processing said image scene in accordance with a predetermined scene transformation requirement; and

a printer for printing out said processed image scene on print media said printer, print media and printing ink stored in a single detachable module inside said camera system;

said camera system comprising a portable hand held unit for the imaging of scenes by said area image sensor and printing said scenes directly out of said camera system via said printer.

2. A camera processor as claimed in claim 1 further comprising a print roll for the storage of print media and printing ink for utilisation by said printer, said print roll being detachable from said camera system.

3. A camera system as claimed in claim 2 wherein said print roll includes an authentication chip containing authentication information and said camera processing means is adapted to interrogate said authentication chip so as to determine the authenticity of said print roll when inserted within said camera system.

4. A camera system as claimed in claim 1 wherein said printer comprises a drop on demand ink printer.

5. A camera system as in claim 1 further comprising a guillotine means for the separation of printed photographs.

6. A camera system as claimed in claim 1 wherein the number of area image sensors is at least 2 and said camera processor means includes means for deriving a stereoscopic image from said area image sensors and said print media includes means for stereoscopic imaging of said stereo images so as to produce a three dimensional affect.

Dated this 15th day of July 1997

Silverbrook Research Pty Ltd
By their Patent Attorneys
GRIFFITH HACK

Abstract

A camera system comprising:

at least one area image sensor for imaging a scene;

a camera processor means for processing said image scene in accordance with a predetermined scene transformation requirement; and

a printer for printing out said processed image scene on print media said printer, print media and printing ink stored in a single detachable module inside said camera system;

said camera system comprising a portable hand held unit for the imaging of scenes by said area image sensor and printing said scenes directly out of said camera system via said printer.

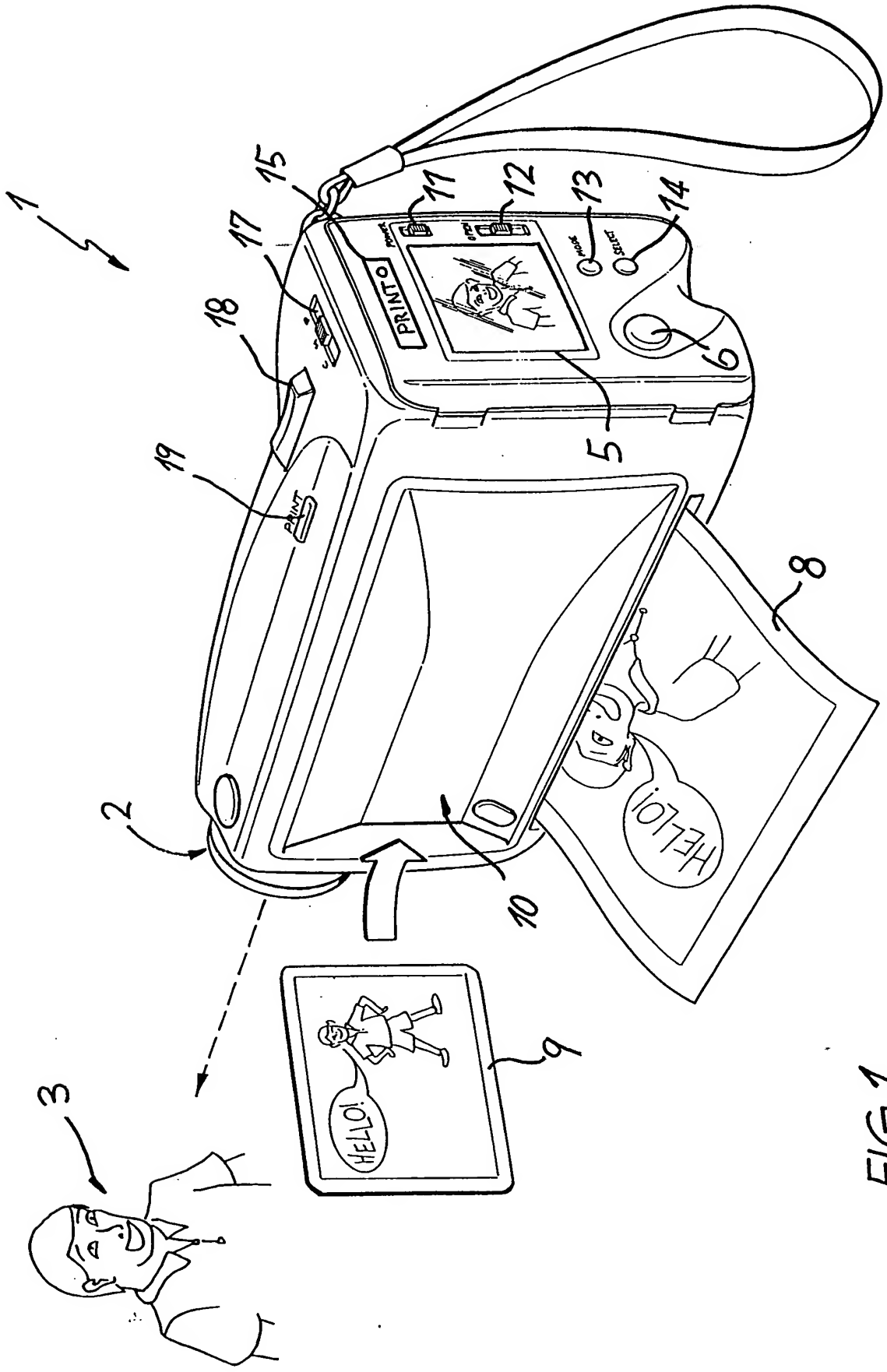



FIG.1

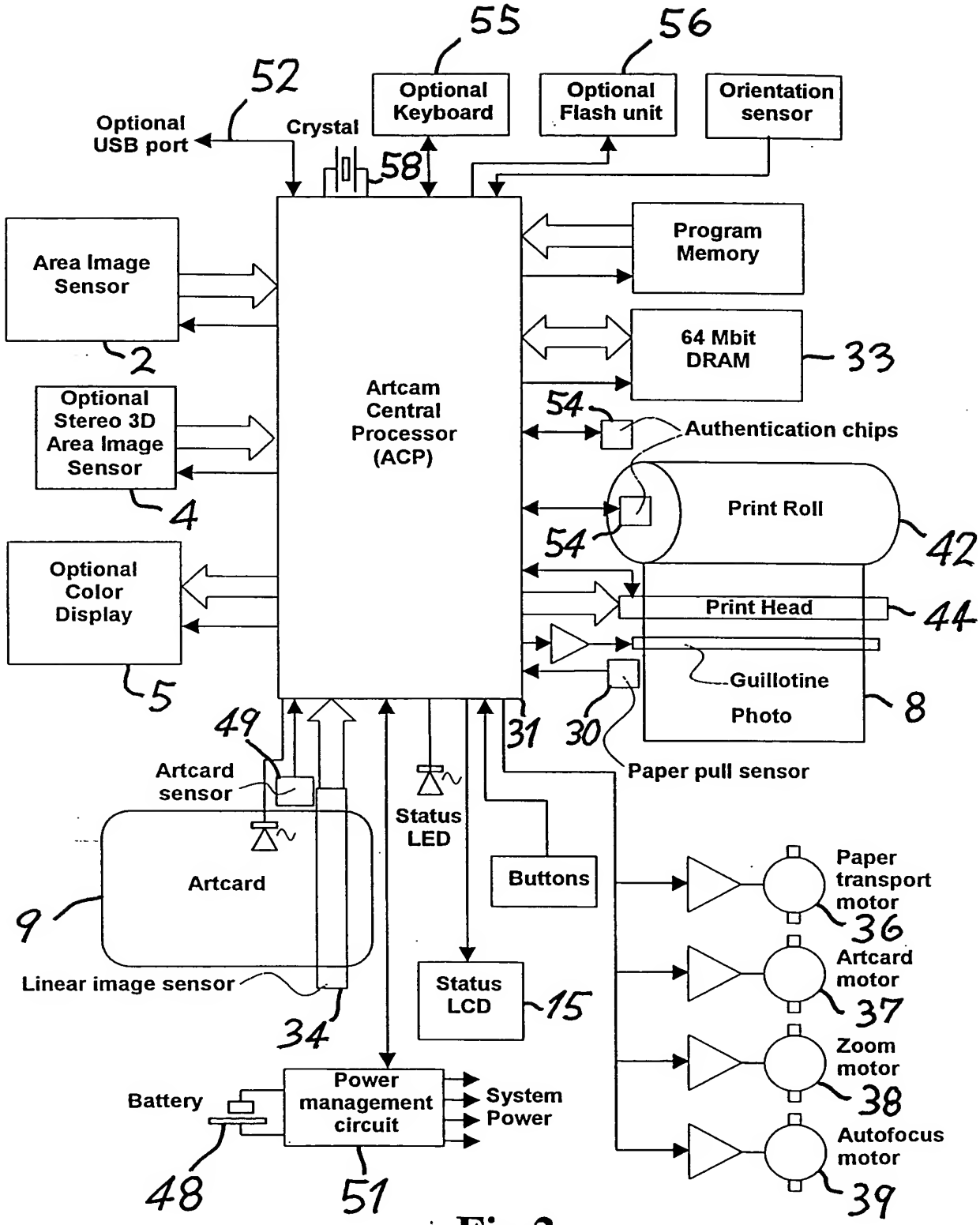

55
56
52
Optional Keyboard
Optional Flash unit
Orientation sensor
Optional USB port
Crystal
58
Area Image Sensor
2
Program Memory
64 Mbit DRAM
33
Optional Stereo 3D Area Image Sensor
4
Artcam Central Processor (ACP)
54
Authentication chips
Print Roll
42
54
Optional Color Display
5
Print Head
44
Guillotine
Photo
8
49
31
30
Paper pull sensor
Artcard sensor
Status LED
Artcard
9
Buttons
Paper transport motor
36
Artcard motor
37
Linear image sensor
34
Status LCD
15
Zoom motor
38
Battery
Power management circuit
System Power
Autofocus motor
48
51
39

Fig 2

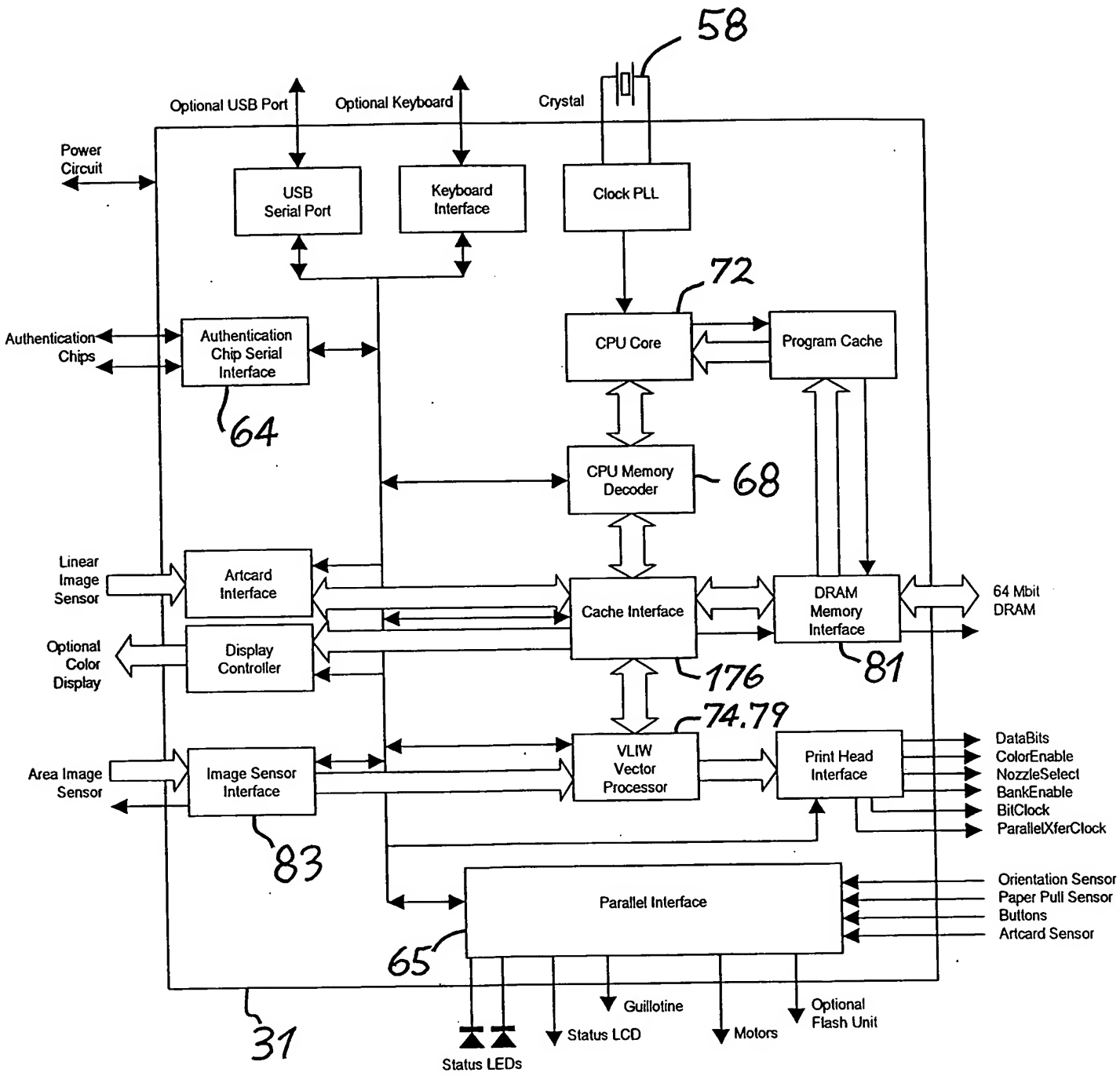
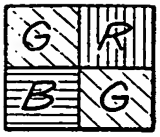

58
Optional USB Port
Optional Keyboard
Crystal
Power Circuit
USB Serial Port
Keyboard Interface
Clock PLL
72
Authentication Chips
Authentication Chip Serial Interface
64
CPU Core
Program Cache
CPU Memory Decoder
68
Linear Image Sensor
Artcard Interface
Cache Interface
DRAM Memory Interface
64 Mbit DRAM
Optional Color Display
Display Controller
176
81
74.79
Area Image Sensor
Image Sensor Interface
VLIW Vector Processor
Print Head Interface
DataBits
ColorEnable
NozzleSelect
BankEnable
BitClock
ParallelXferClock
83
Parallel Interface
Orientation Sensor
Paper Pull Sensor
Buttons
Artcard Sensor
65
31
Guillotine
Optional Flash Unit
Status LCD
Motors
Status LEDs

Fig. 3




2x2 PIXEL BLOCK FROM CCD

**Fig. 4**

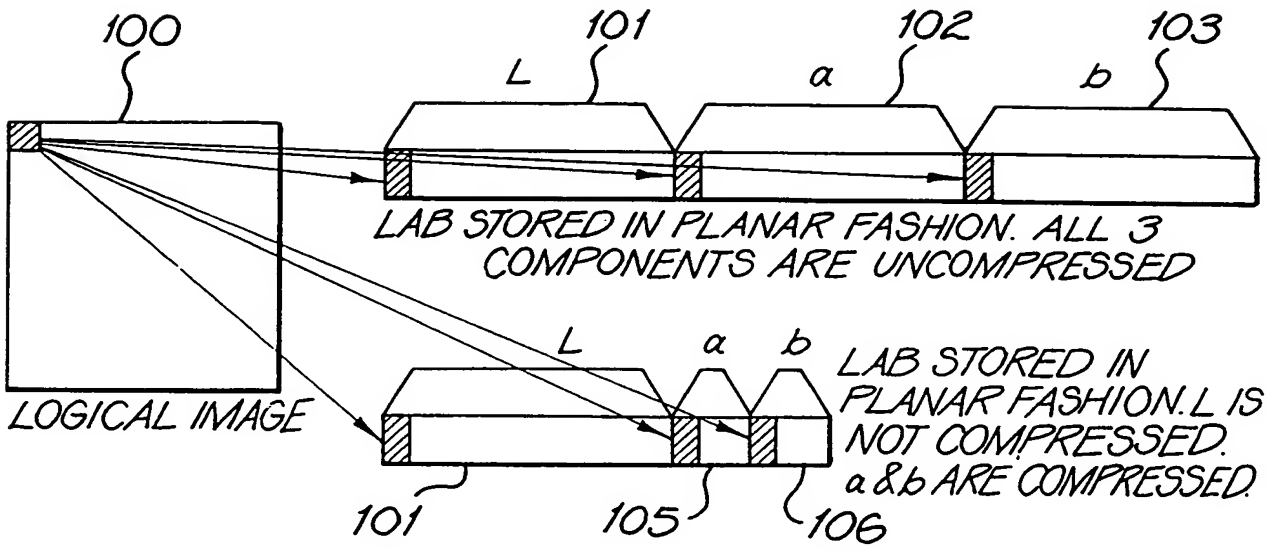



**Fig. 5**

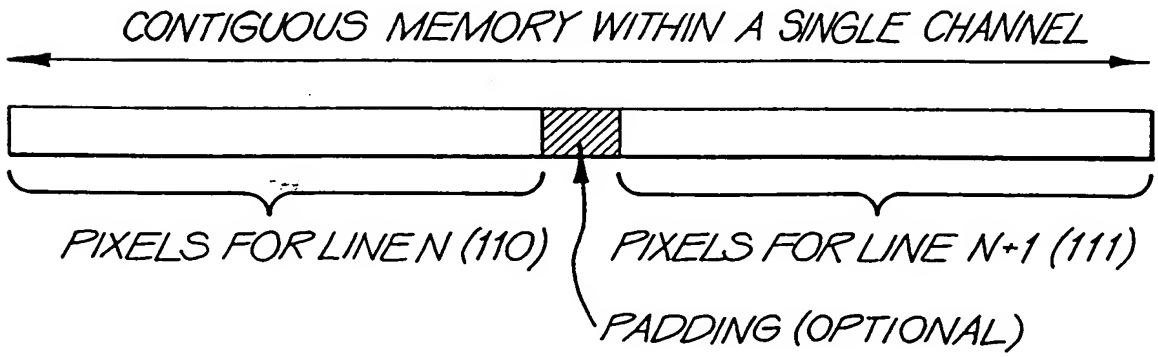



**Fig. 6**

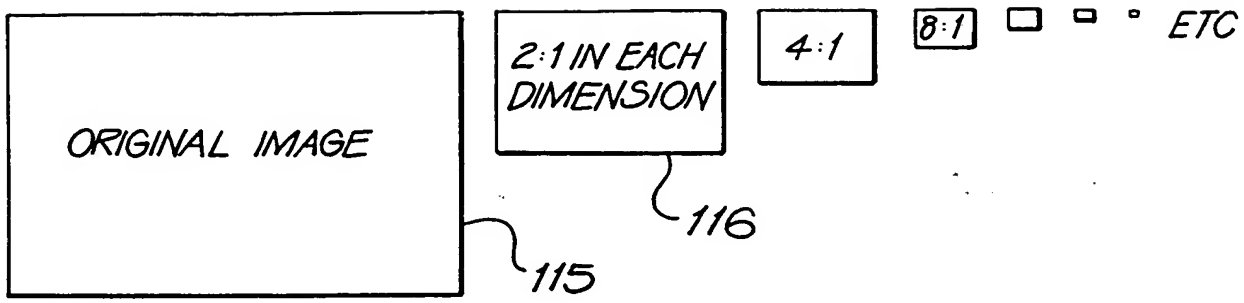

ORIGINAL IMAGE
115
2:1 IN EACH DIMENSION
116
4:1
8:1
ETC

Fig. 7

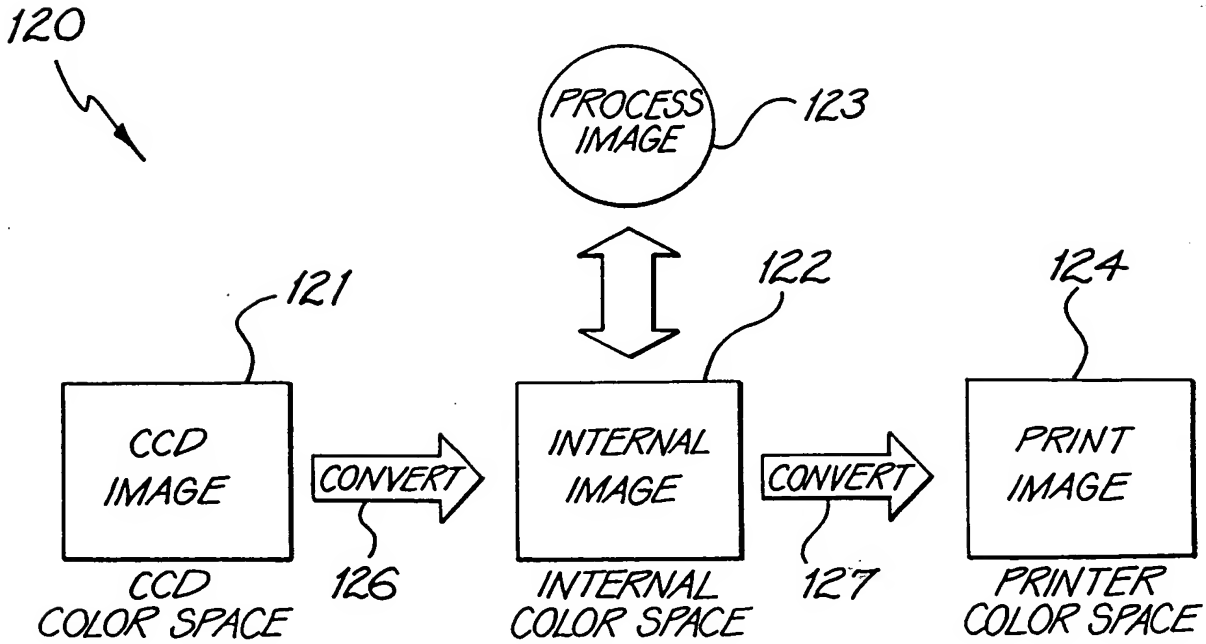

120
PROCESS IMAGE
123
121
122
124
CCD IMAGE
CONVERT
INTERNAL IMAGE
CONVERT
PRINT IMAGE
CCD COLOR SPACE
126
INTERNAL COLOR SPACE
127
PRINTER COLOR SPACE

Fig. 8

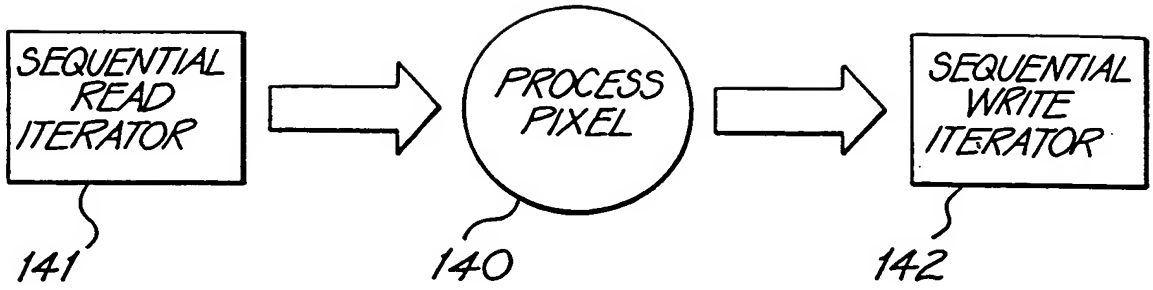

SEQUENTIAL READ ITERATOR
PROCESS PIXEL
SEQUENTIAL WRITE ITERATOR
141
140
142

**Fig. 9**

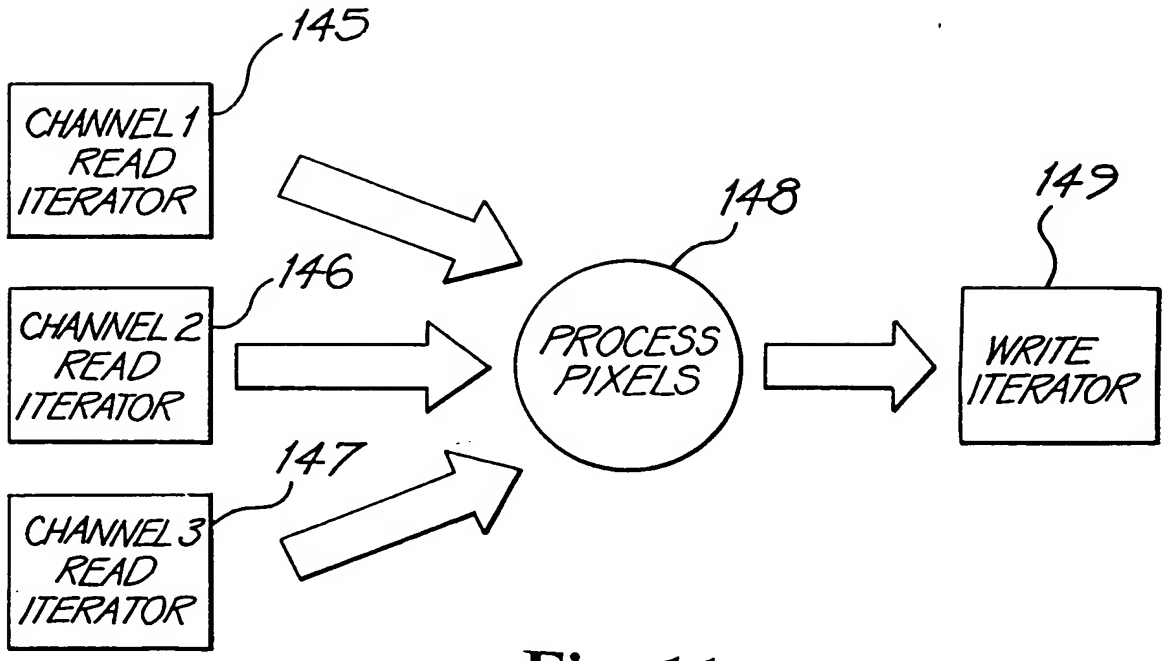

145
CHANNEL 1 READ ITERATOR
146
CHANNEL 2 READ ITERATOR
147
CHANNEL 3 READ ITERATOR
148
PROCESS PIXELS
149
WRITE ITERATOR

**Fig. 11**

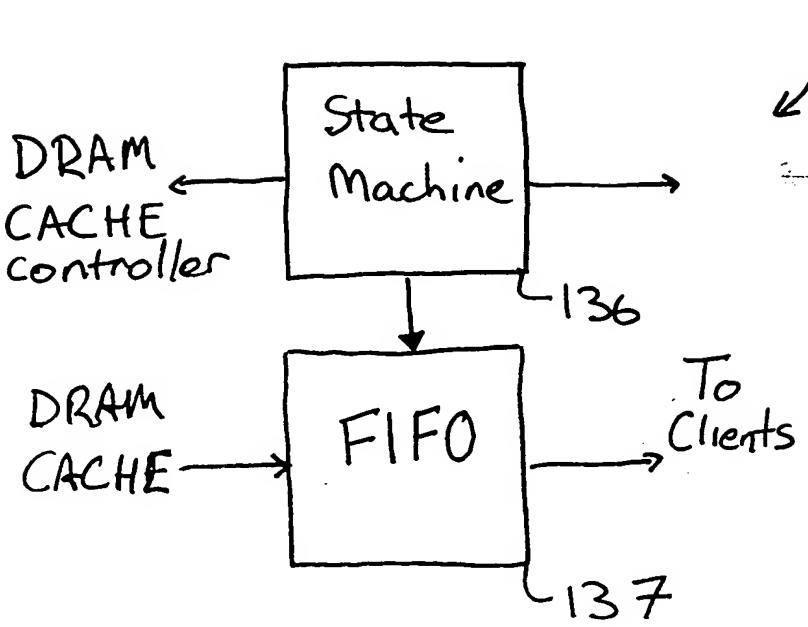

130
State Machine
DRAM CACHE controller
136
DRAM CACHE
FIFO
To Clients
137

Fig. 10

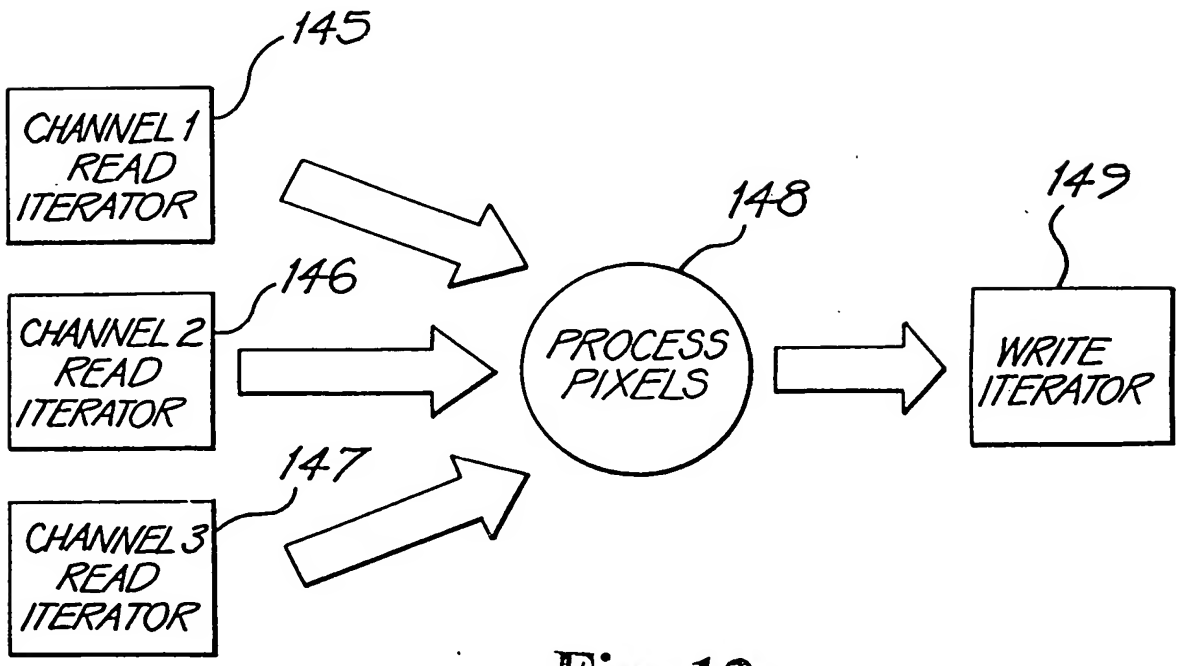

145
CHANNEL 1 READ ITERATOR
146
CHANNEL 2 READ ITERATOR
147
CHANNEL 3 READ ITERATOR
148
PROCESS PIXELS
149
WRITE ITERATOR

Fig. 12

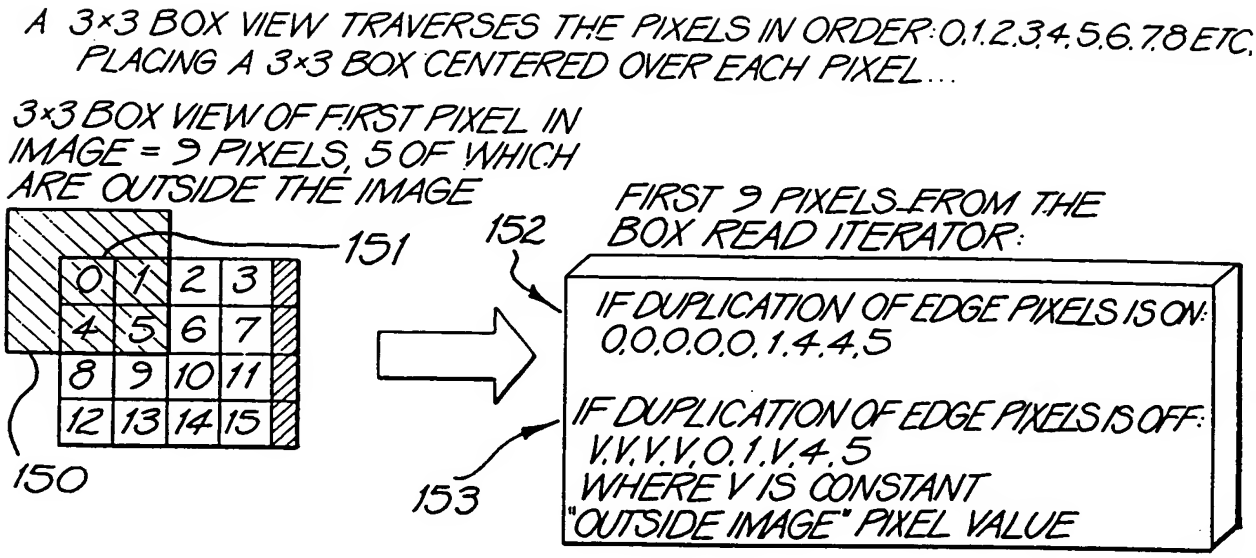

A 3×3 BOX VIEW TRAVERSES THE PIXELS IN ORDER: 0,1,2,3,4,5,6,7,8 ETC.
PLACING A 3×3 BOX CENTERED OVER EACH PIXEL...
3×3 BOX VIEW OF FIRST PIXEL IN IMAGE = 9 PIXELS, 5 OF WHICH ARE OUTSIDE THE IMAGE
151
0 1 2 3
4 5 6 7
8 9 10 11
12 13 14 15
150
152
FIRST 9 PIXELS FROM THE BOX READ ITERATOR:
IF DUPLICATION OF EDGE PIXELS IS ON:
0,0,0,0,0,1,4,4,5
IF DUPLICATION OF EDGE PIXELS IS OFF:
V,V,V,V,0,1,V,4,5
WHERE V IS CONSTANT
"OUTSIDE IMAGE" PIXEL VALUE
153

**Fig. 13**

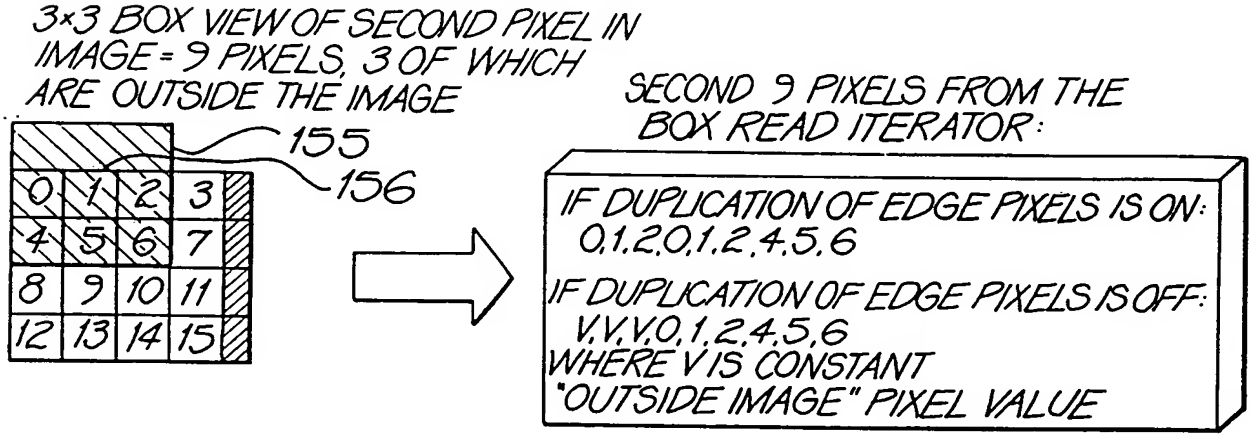

3×3 BOX VIEW OF SECOND PIXEL IN IMAGE = 9 PIXELS, 3 OF WHICH ARE OUTSIDE THE IMAGE
155
156
0 1 2 3
4 5 6 7
8 9 10 11
12 13 14 15
SECOND 9 PIXELS FROM THE BOX READ ITERATOR:
IF DUPLICATION OF EDGE PIXELS IS ON:
0,1,2,0,1,2,4,5,6
IF DUPLICATION OF EDGE PIXELS IS OFF:
V,V,V,0,1,2,4,5,6
WHERE V IS CONSTANT
"OUTSIDE IMAGE" PIXEL VALUE

**Fig. 14**

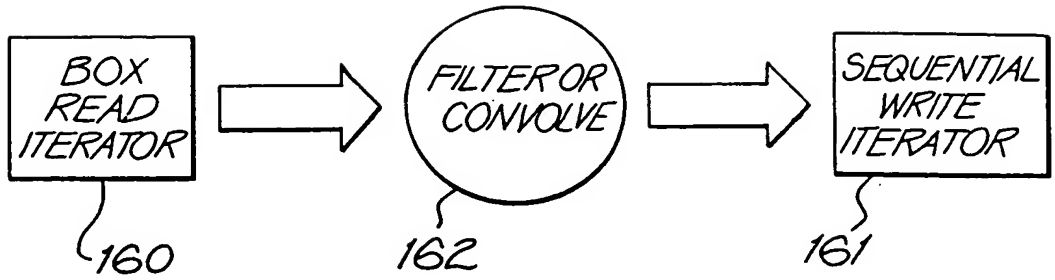

BOX READ ITERATOR
FILTER OR CONVOLVE
SEQUENTIAL WRITE ITERATOR
160
162
161

Fig. 15

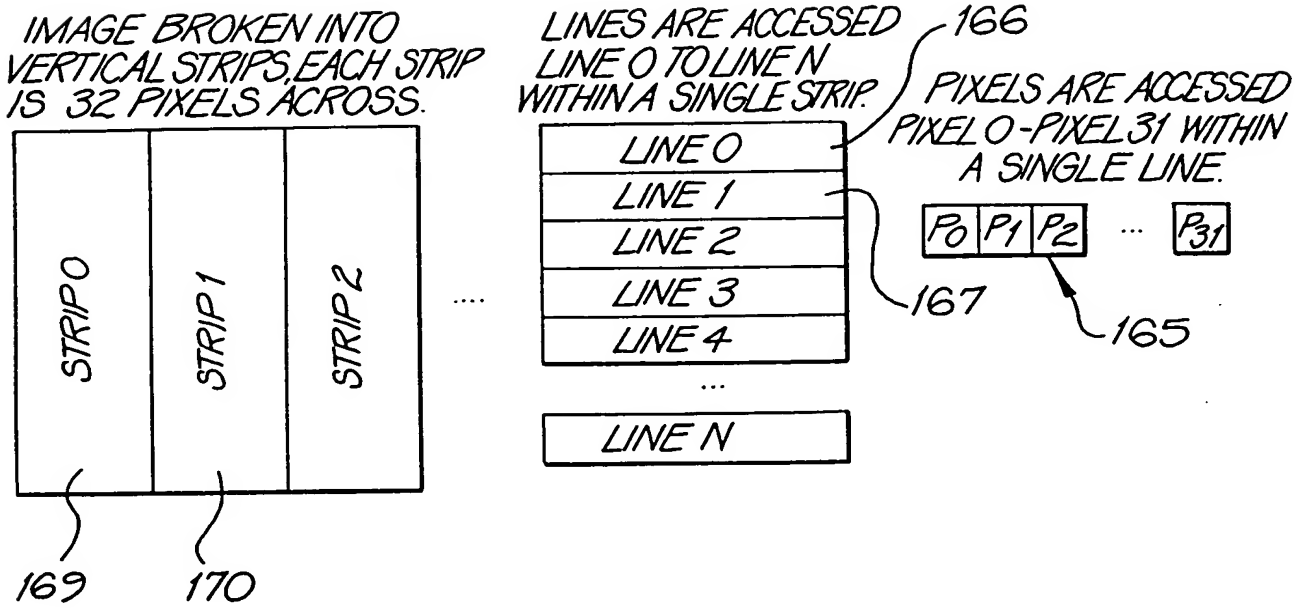


IMAGE BROKEN INTO VERTICAL STRIPS. EACH STRIP IS 32 PIXELS ACROSS.
STRIP 0
STRIP 1
STRIP 2
169
170
LINES ARE ACCESSED LINE 0 TO LINE N WITHIN A SINGLE STRIP.
166
LINE 0
LINE 1
LINE 2
LINE 3
LINE 4
LINE N
167
PIXELS ARE ACCESSED PIXEL 0 - PIXEL 31 WITHIN A SINGLE LINE.
P0 P1 P2 ... P31
165

Fig. 16

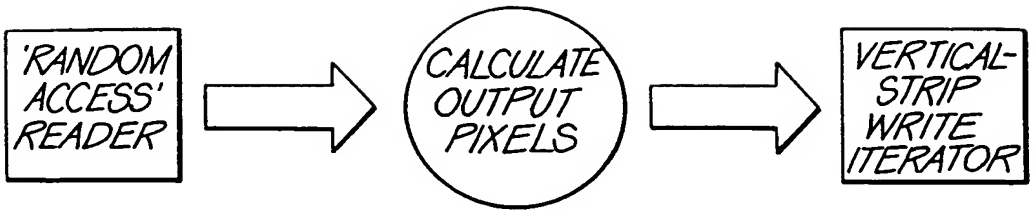

'RANDOM ACCESS' READER
CALCULATE OUTPUT PIXELS
VERTICAL-STRIP WRITE ITERATOR

Fig. 17

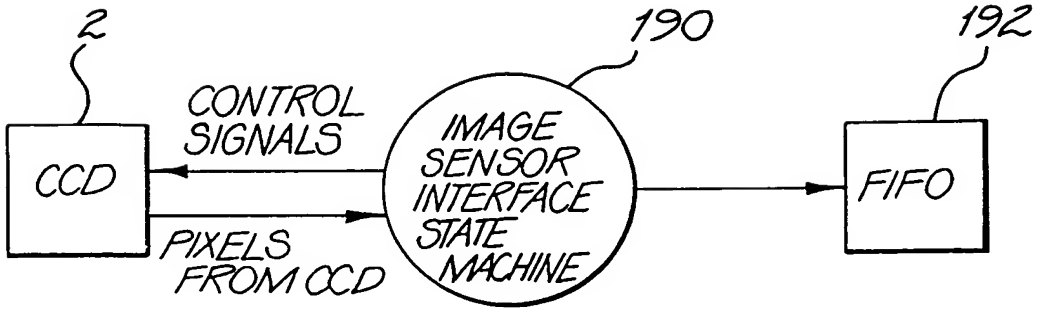

2
190
192
CONTROL SIGNALS
CCD
IMAGE SENSOR INTERFACE STATE MACHINE
FIFO
PIXELS FROM CCD

Fig. 25

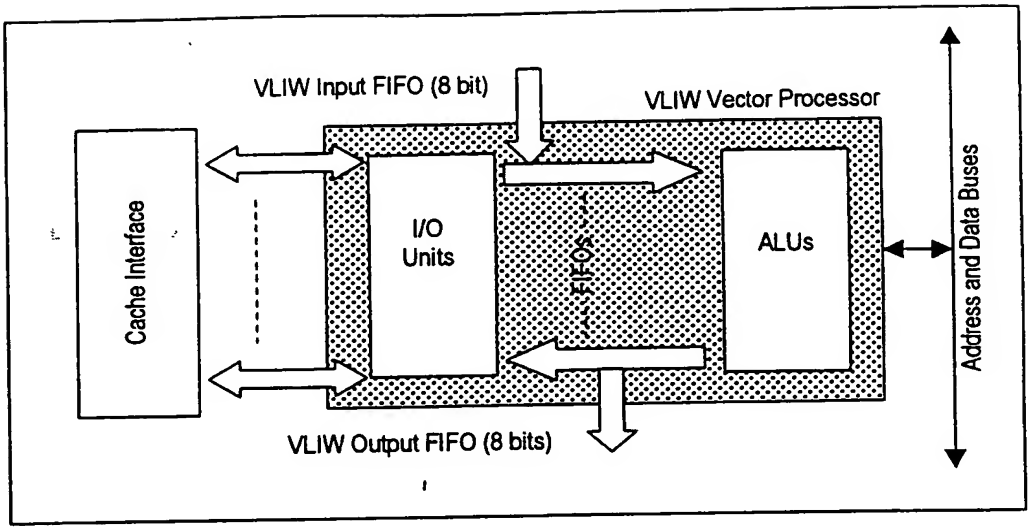

VLIW Input FIFO (8 bit)
VLIW Vector Processor
Cache Interface
I/O Units
FIFOs
ALUs
Address and Data Buses
VLIW Output FIFO (8 bits)

Fig. 18

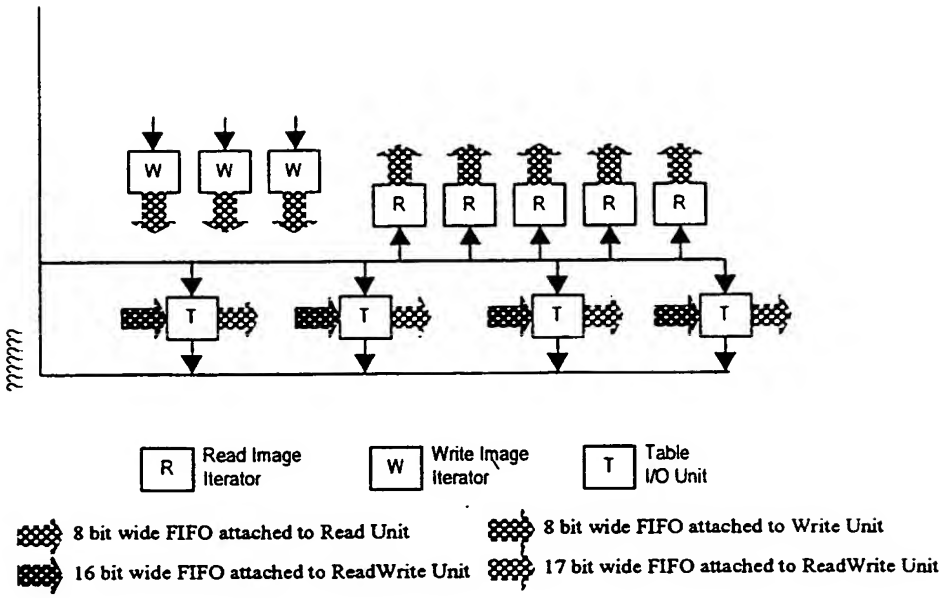

W
W
W
R
R
R
R
R
T
T
T
T
R
Read Image Iterator
W
Write Image Iterator
T
Table I/O Unit
8 bit wide FIFO attached to Read Unit
16 bit wide FIFO attached to ReadWrite Unit
8 bit wide FIFO attached to Write Unit
17 bit wide FIFO attached to ReadWrite Unit

**Fig. 19**

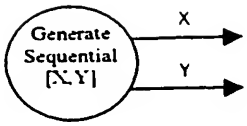

Generate
Sequential
[X,Y]
X
Y

Fig. 20

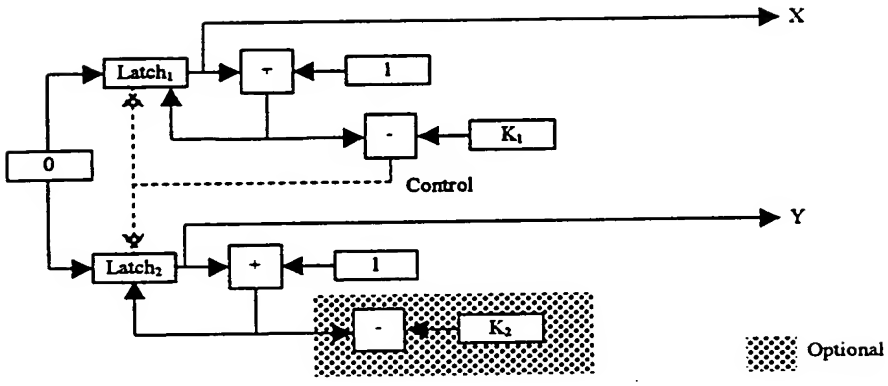

X
Latch1
-
1
-
K1
0
Control
Y
Latch2
+
1
-
K2
Optional

Fig. 21

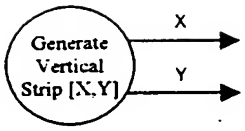

Generate
Vertical
Strip [X,Y]
X
Y

Fig. 22

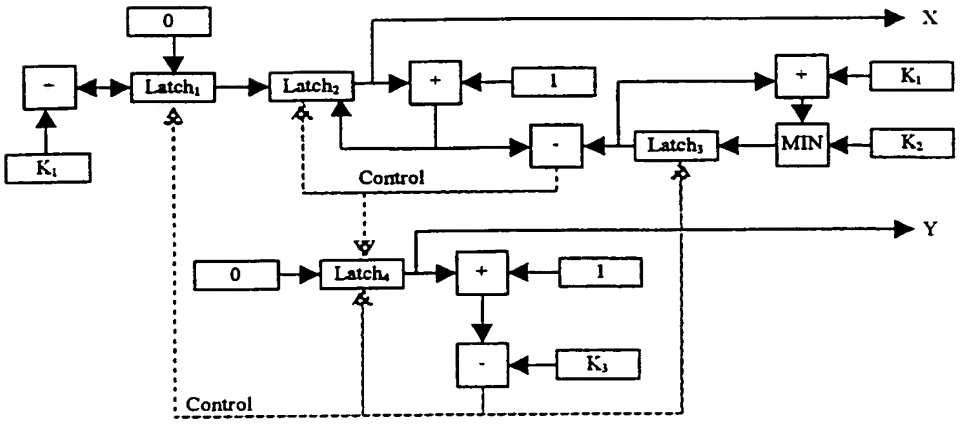

0
Latch1
Latch2
+
1
-
K1
Control
X
+
K1
MIN
K2
Latch3
Y
0
Latch4
+
1
-
K3
Control

Fig. 23

2x2 pixel block from CCD

Fig. 24

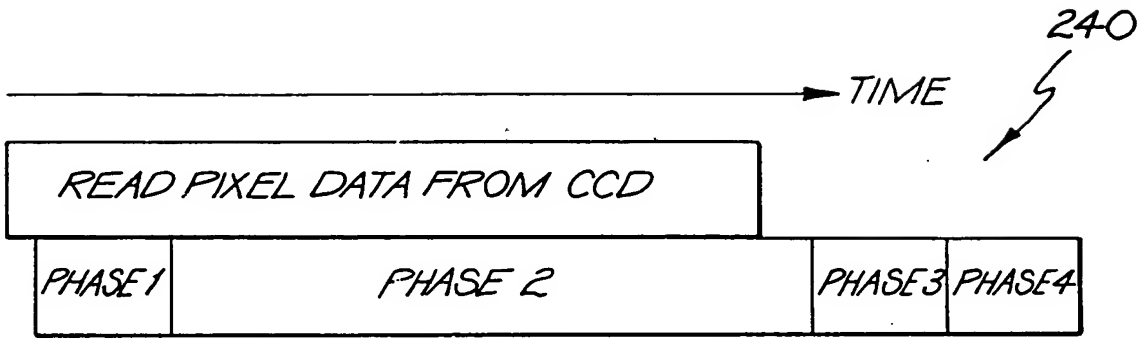

240
TIME
READ PIXEL DATA FROM CCD
PHASE 1
PHASE 2
PHASE 3
PHASE 4

Fig. 29

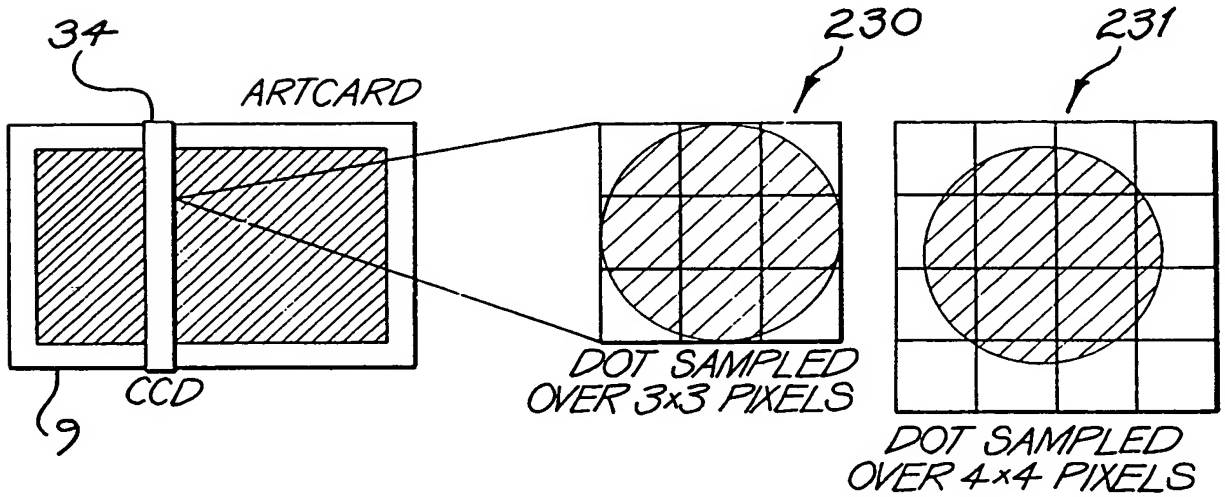

34
ARTCARD
230
231
CCD
9
DOT SAMPLED
OVER 3x3 PIXELS
DOT SAMPLED
OVER 4x4 PIXELS

Fig. 26

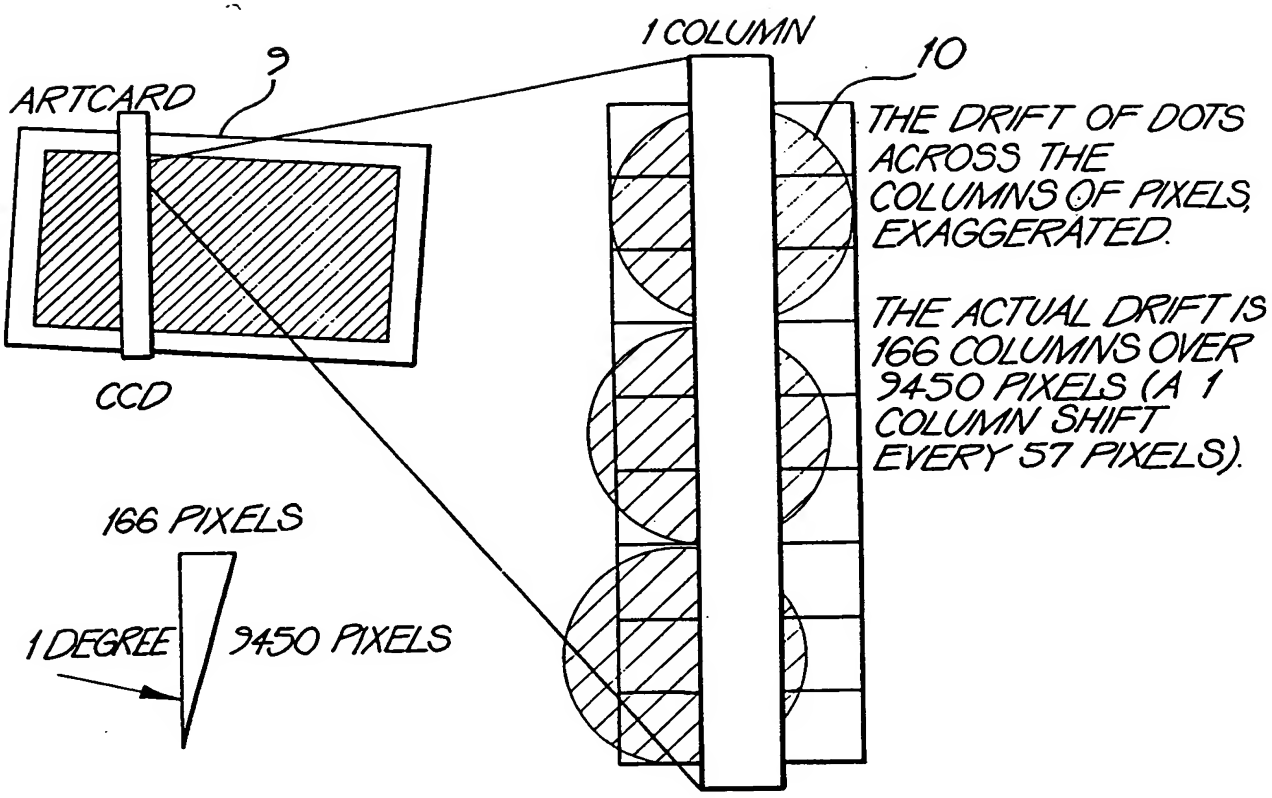

1 COLUMN
10
ARTCARD
9
THE DRIFT OF DOTS ACROSS THE COLUMNS OF PIXELS, EXAGGERATED.
THE ACTUAL DRIFT IS 166 COLUMNS OVER 9450 PIXELS (A 1 COLUMN SHIFT EVERY 57 PIXELS).
CCD
166 PIXELS
1 DEGREE
9450 PIXELS

Fig. 27

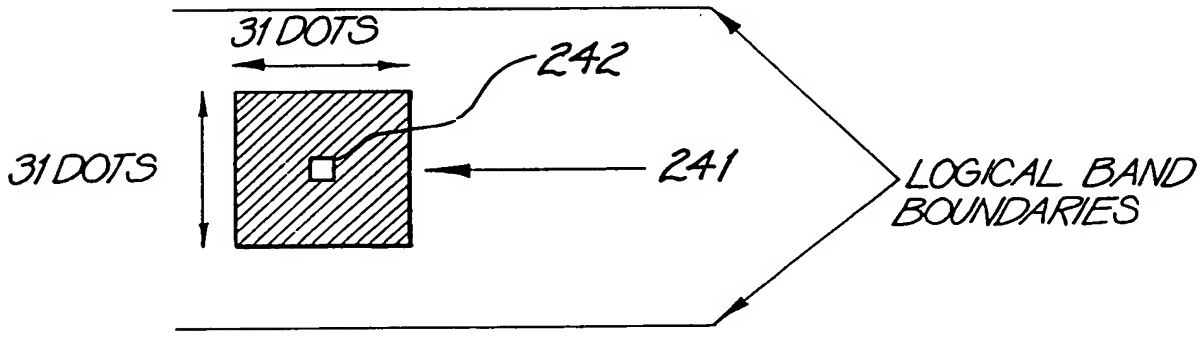

31 DOTS
242
31 DOTS
241
LOGICAL BAND BOUNDARIES

Fig. 31

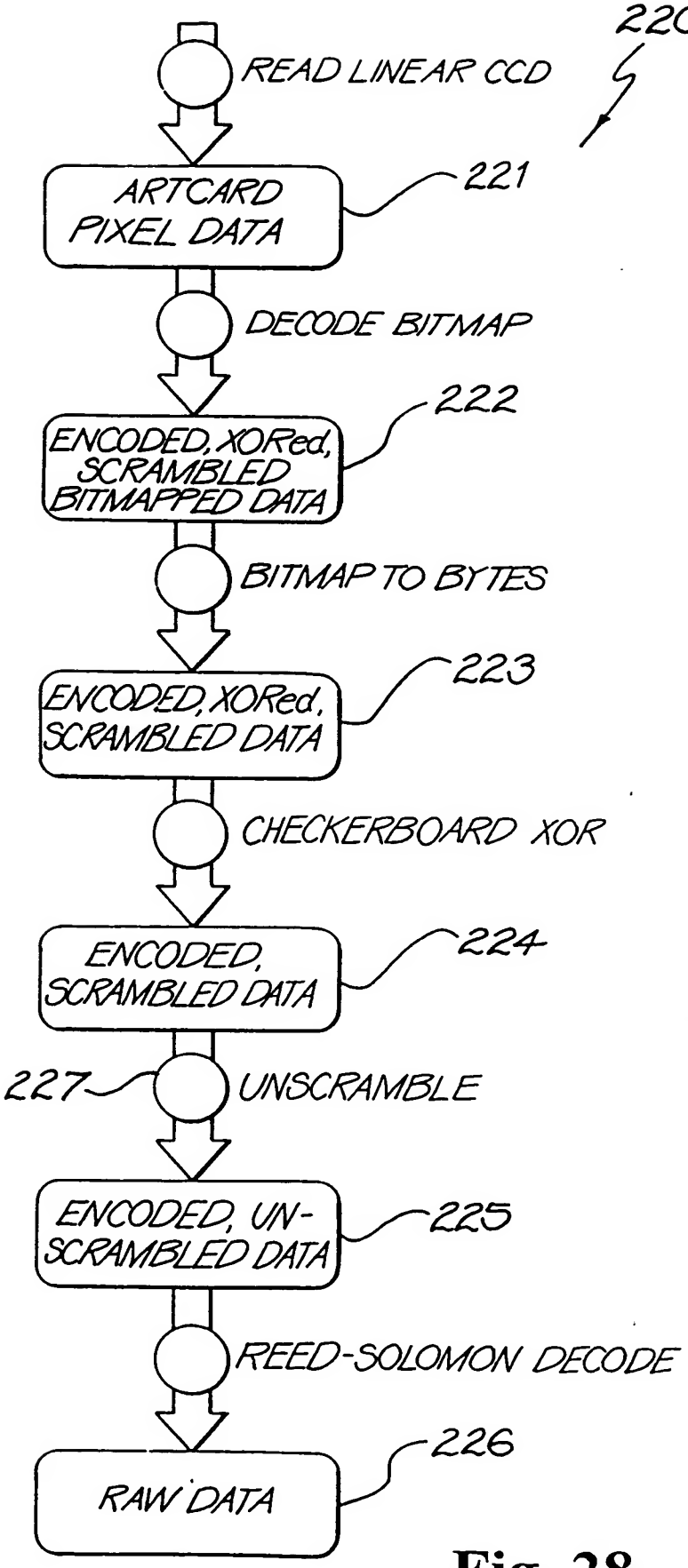

220
READ LINEAR CCD
ARTCARD PIXEL DATA
221
DECODE BITMAP
ENCODED, XORed, SCRAMBLED BITMAPPED DATA
222
BITMAP TO BYTES
ENCODED, XORed, SCRAMBLED DATA
223
CHECKERBOARD XOR
ENCODED, SCRAMBLED DATA
224
227
UNSCRAMBLE
ENCODED, UN-SCRAMBLED DATA
225
REED-SOLOMON DECODE
RAW DATA
226

**Fig. 28**

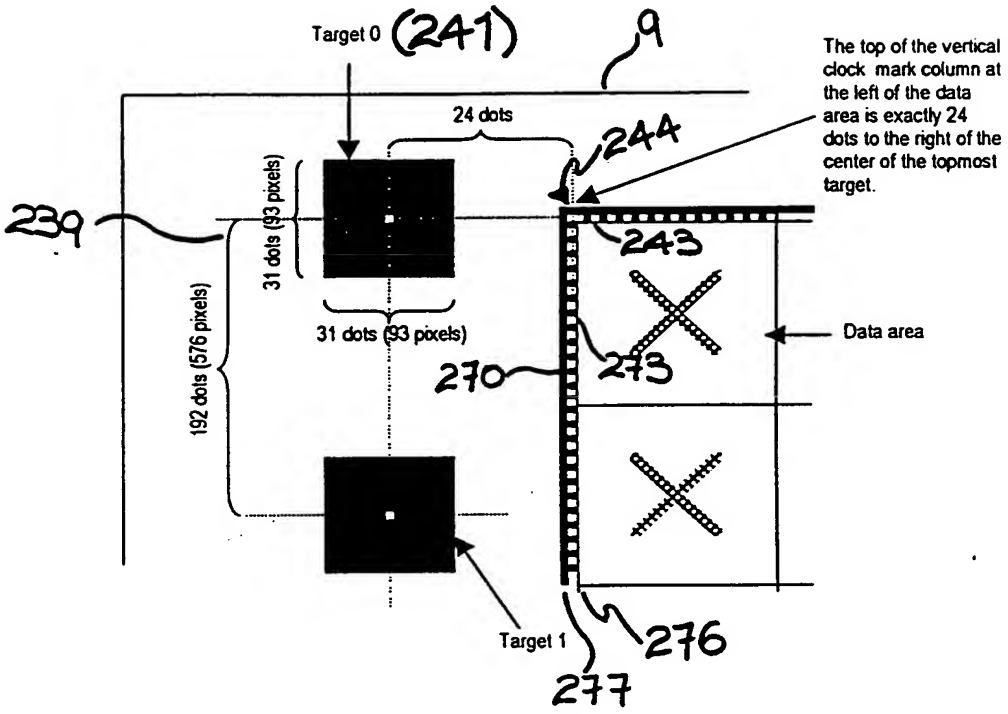

Target 0 (241)
9
The top of the vertical clock mark column at the left of the data area is exactly 24 dots to the right of the center of the topmost target.
24 dots
244
239
31 dots (93 pixels)
243
31 dots (93 pixels)
Data area
192 dots (576 pixels)
270
273
Target 1
276
277

Fig. 30

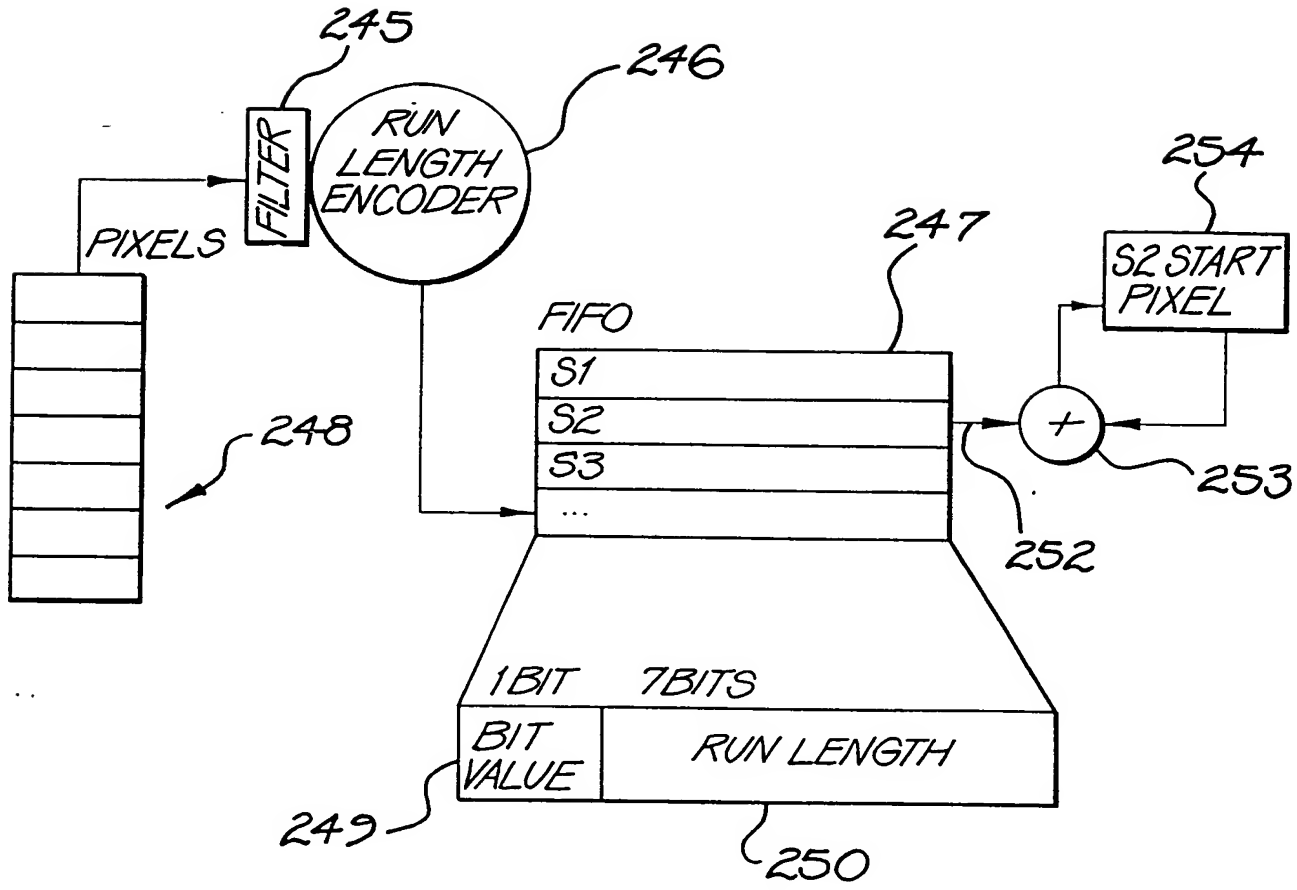

245
246
FILTER
RUN LENGTH ENCODER
PIXELS
254
247
S2 START PIXEL
FIFO
S1
S2
S3
...
248
+
253
252
1BIT
7BITS
BIT VALUE
RUN LENGTH
249
250

**Fig. 32**

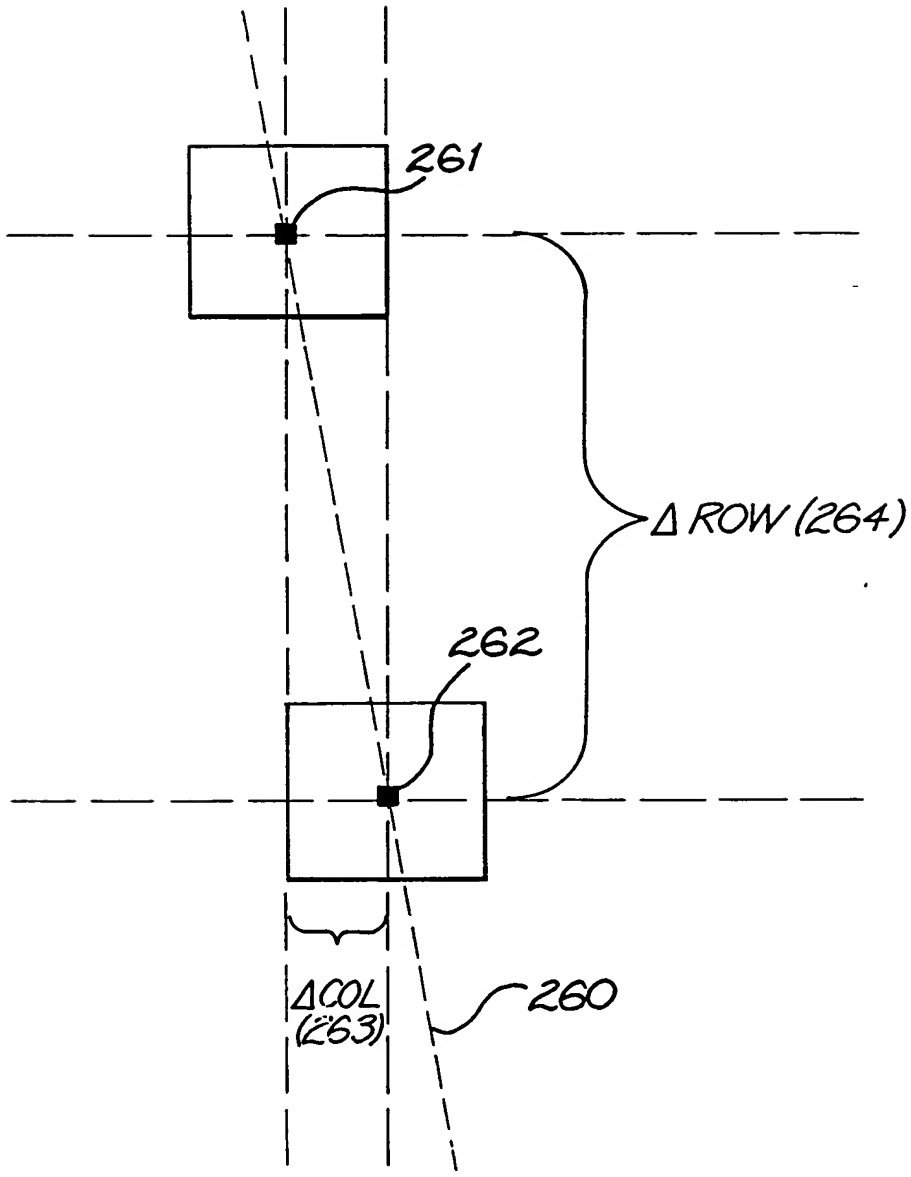



Fig. 33

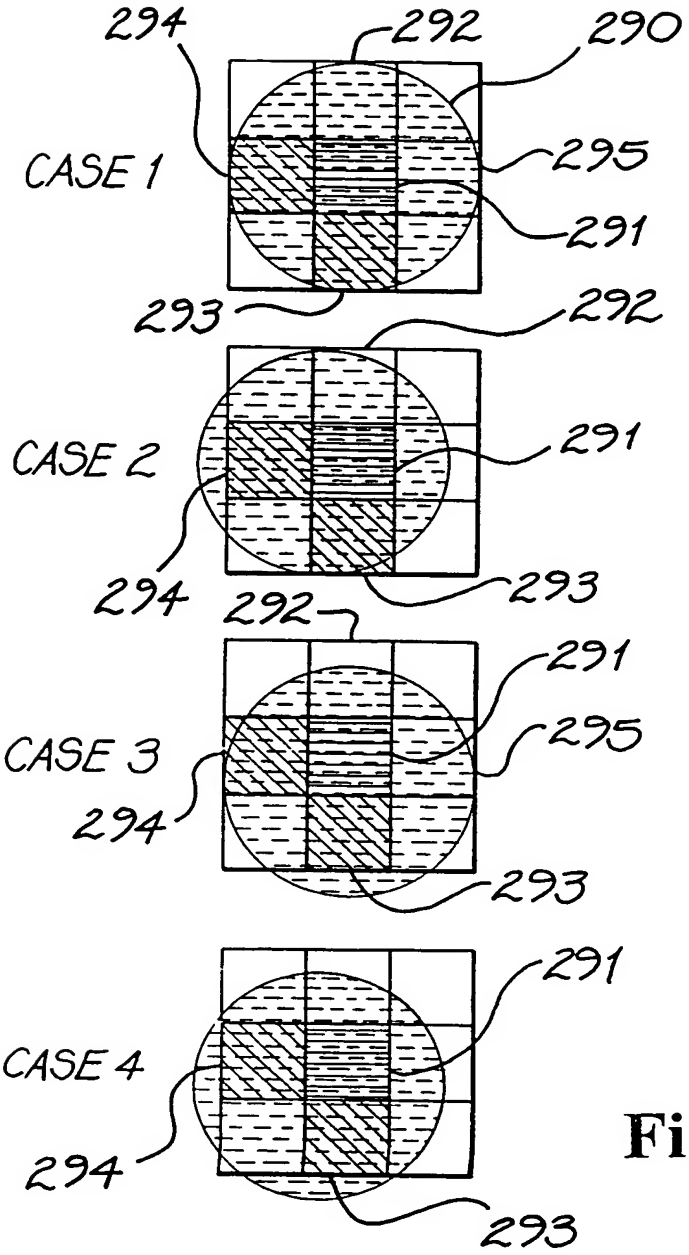

294
292
290
CASE 1
295
291
293
292
CASE 2
291
294
293
292
291
CASE 3
295
294
293
291
CASE 4
294
293

Fig. 34

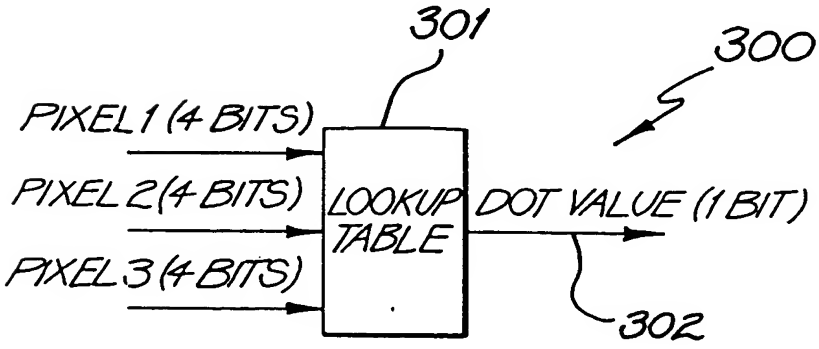

301
300
PIXEL 1 (4 BITS)
PIXEL 2 (4 BITS)
PIXEL 3 (4 BITS)
LOOKUP TABLE
DOT VALUE (1 BIT)
302

Fig. 35

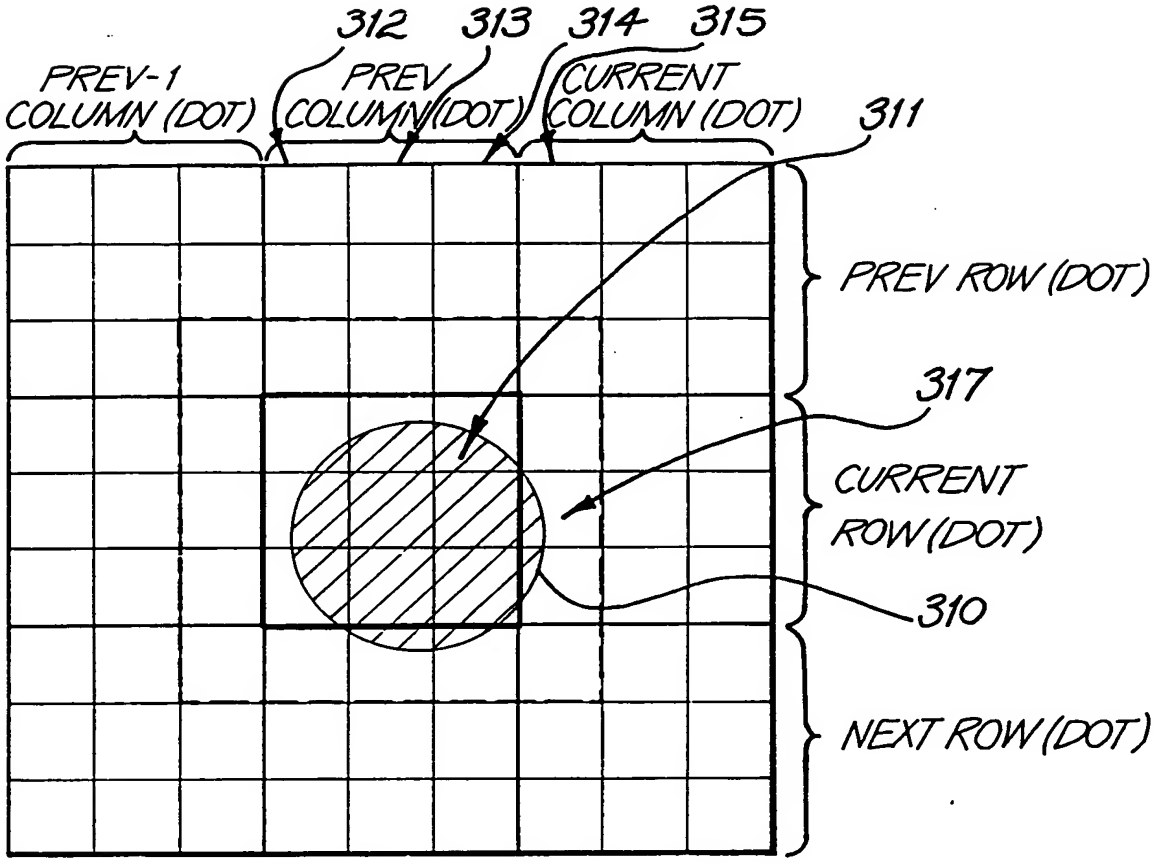

312
313
314
315
PREV-1 COLUMN (DOT)
PREV COLUMN (DOT)
CURRENT COLUMN (DOT)
311
PREV ROW (DOT)
317
CURRENT ROW (DOT)
310
NEXT ROW (DOT)

Fig. 36

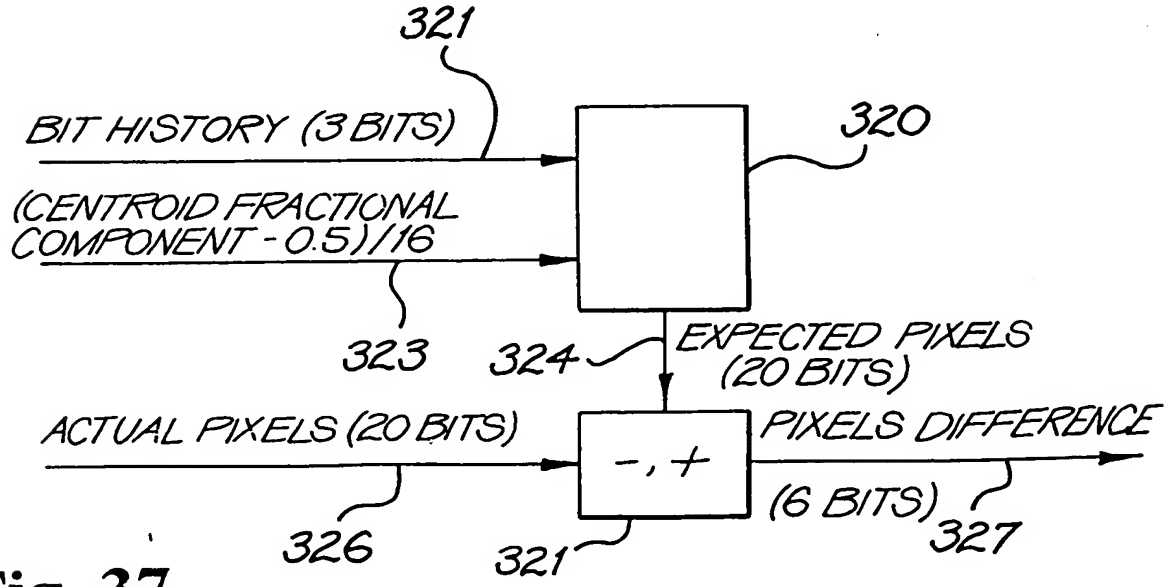

321
BIT HISTORY (3 BITS)
320
(CENTROID FRACTIONAL COMPONENT - 0.5)/16
323
324
EXPECTED PIXELS (20 BITS)
ACTUAL PIXELS (20 BITS)
-, +
PIXELS DIFFERENCE
(6 BITS)
326
321
327

Fig. 37

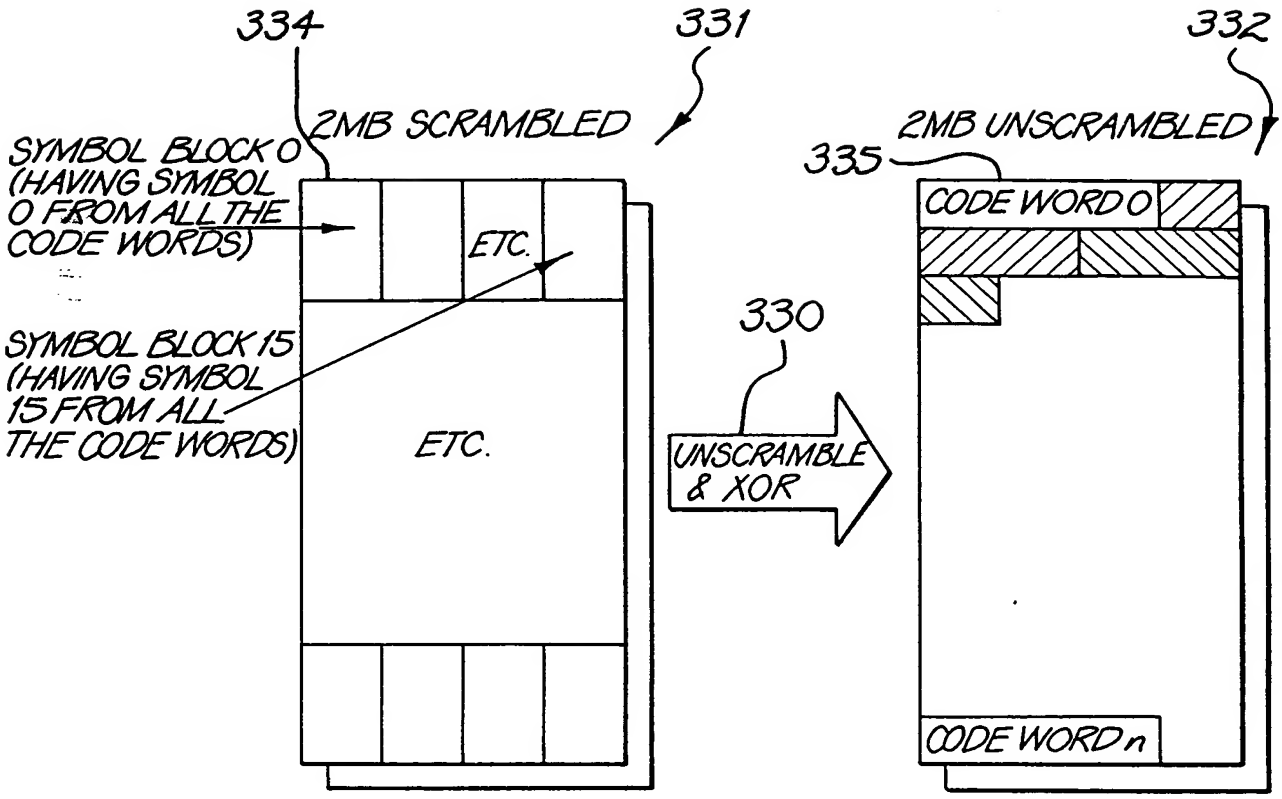

334
331
332
2MB SCRAMBLED
2MB UNSCRAMBLED
335
SYMBOL BLOCK 0
(HAVING SYMBOL
0 FROM ALL THE
CODE WORDS)
ETC.
SYMBOL BLOCK 15
(HAVING SYMBOL
15 FROM ALL
THE CODE WORDS)
ETC.
330
UNSCRAMBLE
& XOR
CODE WORD 0
CODE WORD n

**Fig. 38**

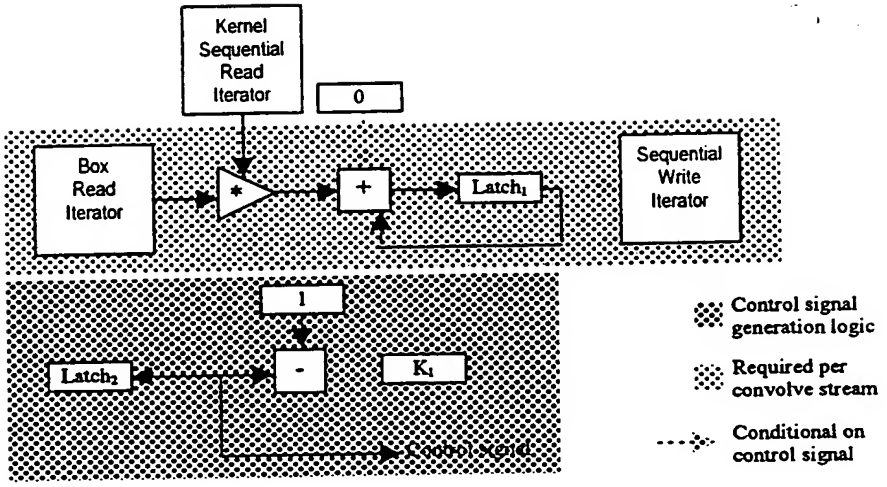

Kernel Sequential Read Iterator
0
Box Read Iterator
*
+
Latch1
Sequential Write Iterator
1
Latch2
-
K1
Control signal generation logic
Required per convolve stream
Conditional on control signal

Fig. 40

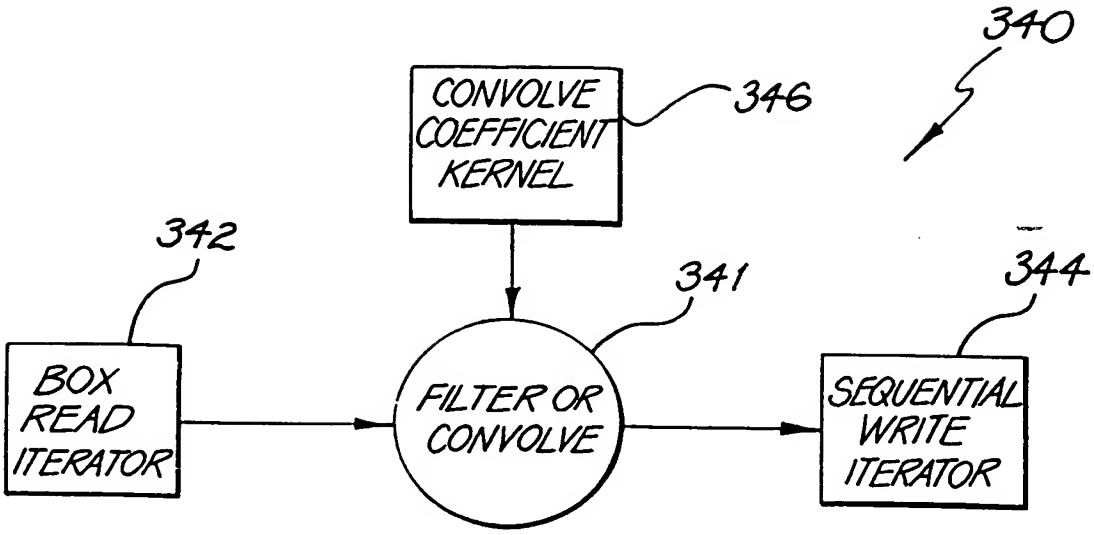

340
CONVOLVE COEFFICIENT KERNEL
346
342
341
344
BOX READ ITERATOR
FILTER OR CONVOLVE
SEQUENTIAL WRITE ITERATOR

Fig. 39

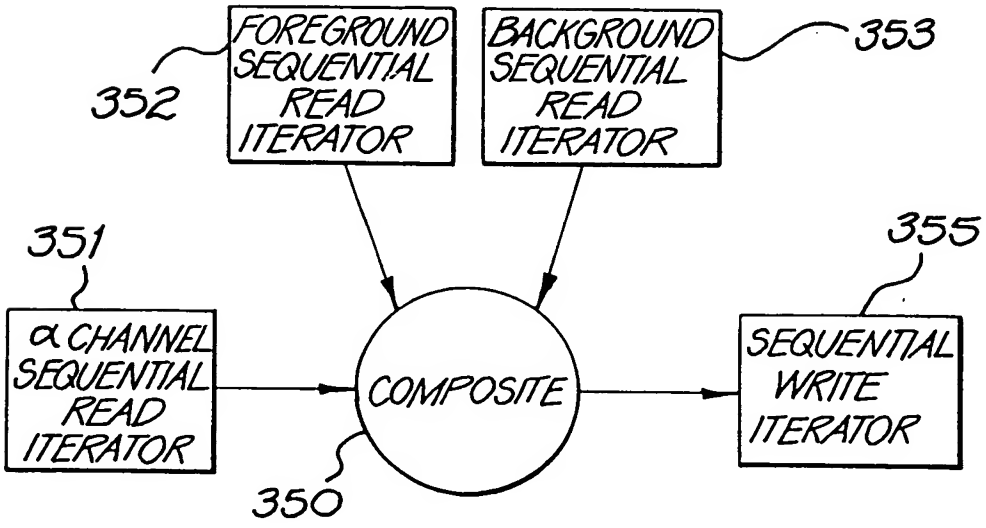

FOREGROUND SEQUENTIAL READ ITERATOR
BACKGROUND SEQUENTIAL READ ITERATOR
352
353
351
355
α CHANNEL SEQUENTIAL READ ITERATOR
COMPOSITE
SEQUENTIAL WRITE ITERATOR
350

Fig. 41

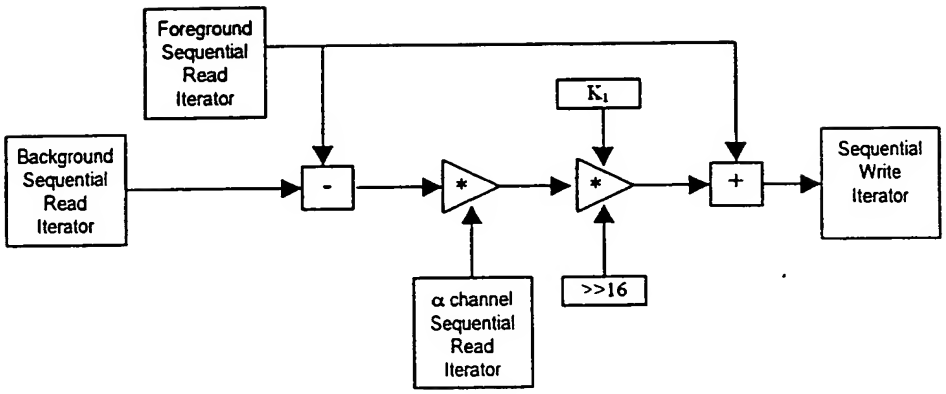

Foreground Sequential Read Iterator
Background Sequential Read Iterator
-
*
K1
*
+
Sequential Write Iterator
α channel Sequential Read Iterator
>>16

Fig. 42

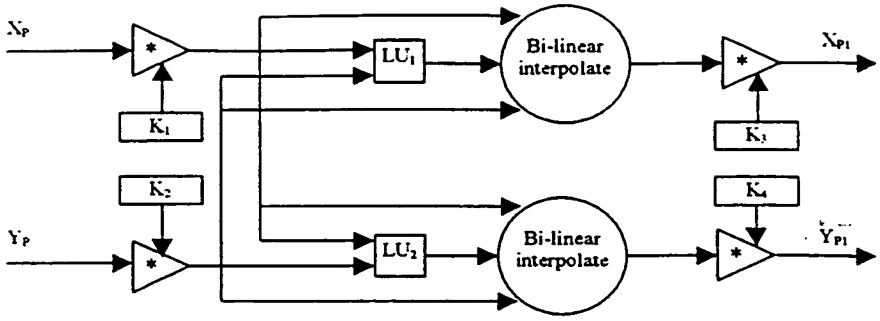

$X_P$
$Y_P$
$K_1$
$K_2$
$LU_1$
$LU_2$
Bi-linear interpolate
Bi-linear interpolate
$K_3$
$K_4$
$X_{P1}$
$Y_{P1}$

Fig. 44

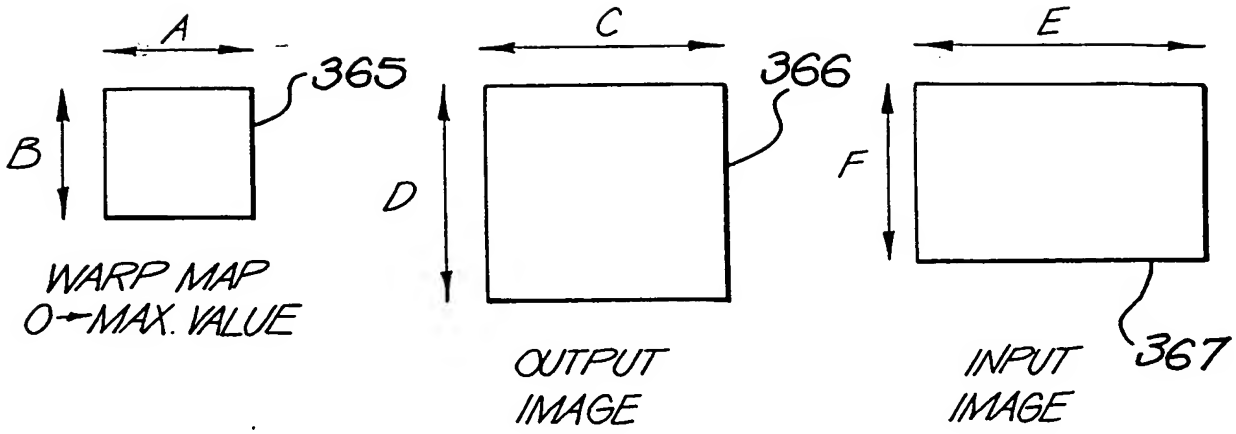

A
365
B
WARP MAP
0→MAX. VALUE
C
366
D
OUTPUT
IMAGE
E
F
INPUT
IMAGE
367

Fig. 43

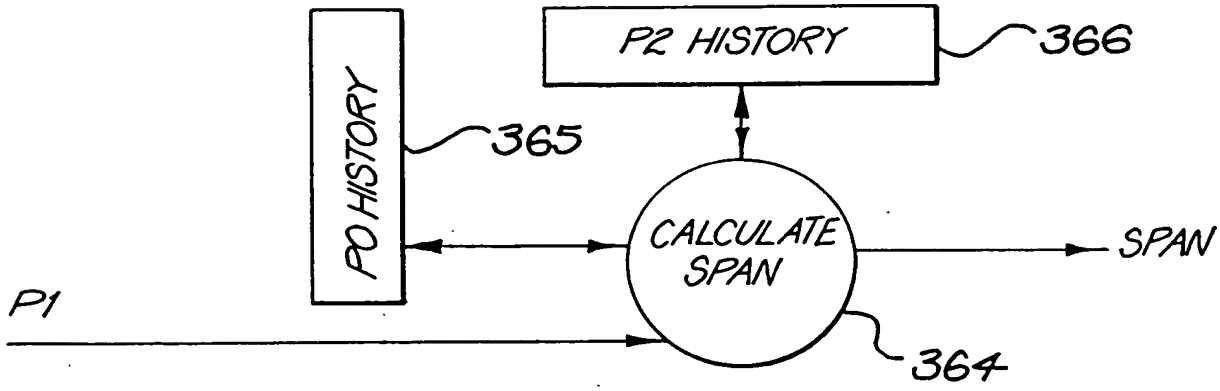

P2 HISTORY
366
P0 HISTORY
365
CALCULATE
SPAN
SPAN
P1
364

Fig. 46

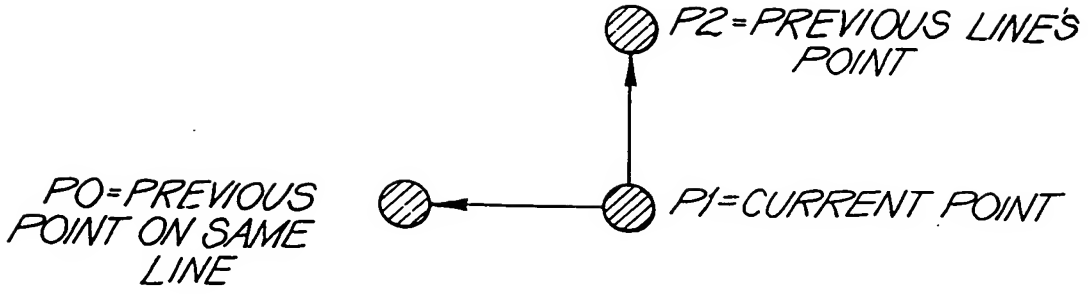

P2=PREVIOUS LINE'S POINT
P0=PREVIOUS POINT ON SAME LINE
P1=CURRENT POINT

Fig. 45

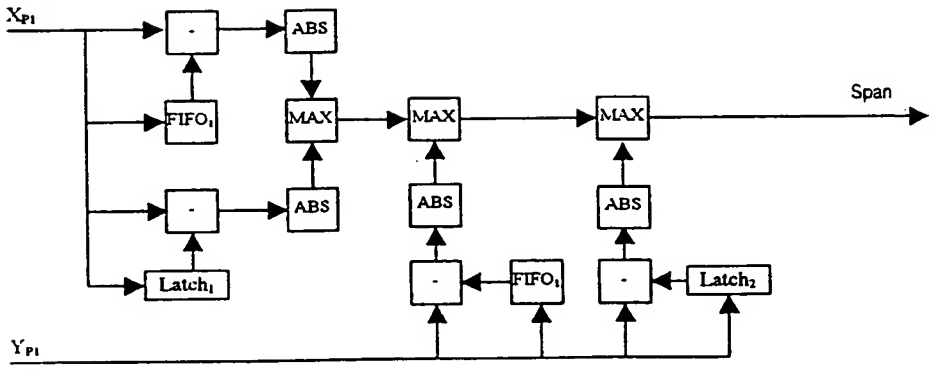

$X_{P1}$
-
ABS
FIFO1
MAX
MAX
MAX
Span
-
ABS
ABS
ABS
Latch1
-
FIFO1
-
Latch2
$Y_{P1}$

Fig. 47

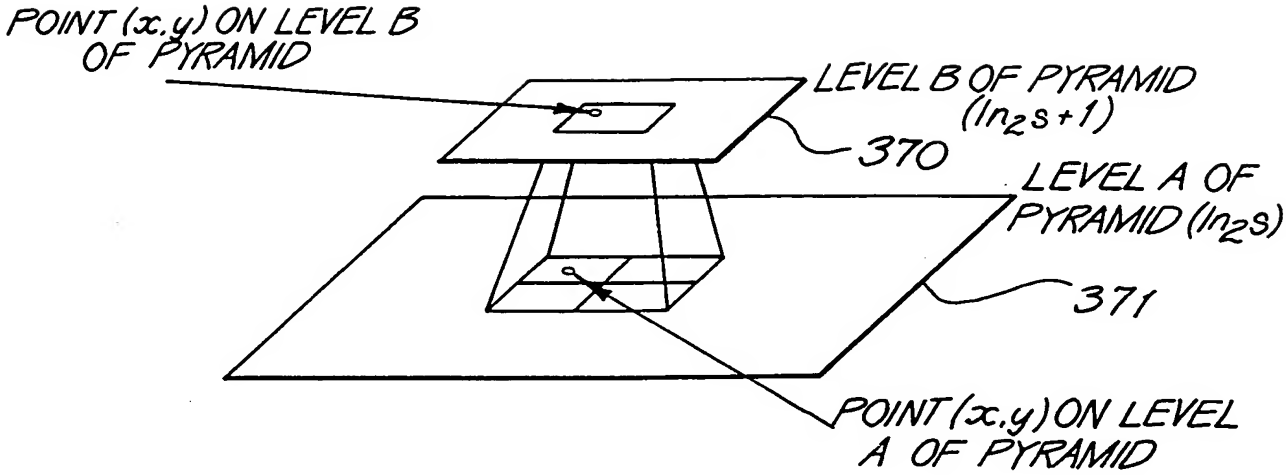

POINT (x,y) ON LEVEL B OF PYRAMID
LEVEL B OF PYRAMID ($ln_2 s+1$)
370
LEVEL A OF PYRAMID ($ln_2 s$)
371
POINT (x,y) ON LEVEL A OF PYRAMID

**Fig. 48**

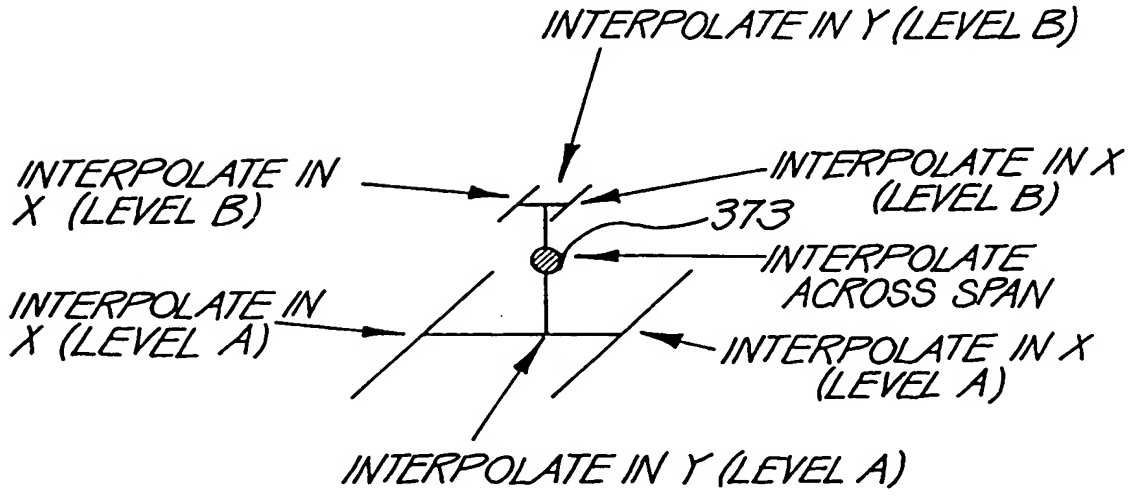

INTERPOLATE IN Y (LEVEL B)
INTERPOLATE IN X (LEVEL B)
INTERPOLATE IN X (LEVEL B)
373
INTERPOLATE ACROSS SPAN
INTERPOLATE IN X (LEVEL A)
INTERPOLATE IN X (LEVEL A)
INTERPOLATE IN Y (LEVEL A)

**Fig. 49**

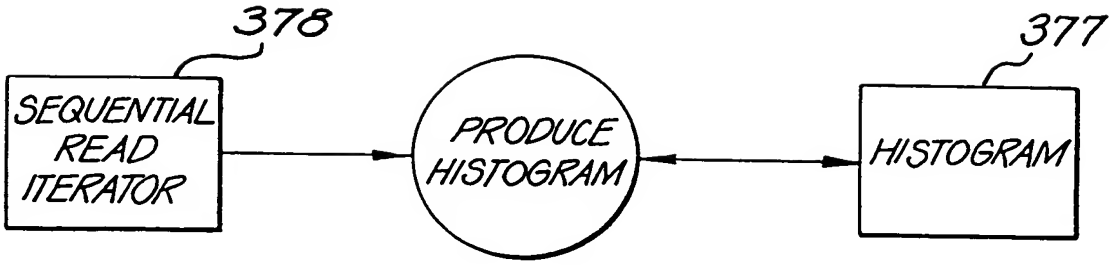

378
377
SEQUENTIAL READ ITERATOR
PRODUCE HISTOGRAM
HISTOGRAM

Fig. 50

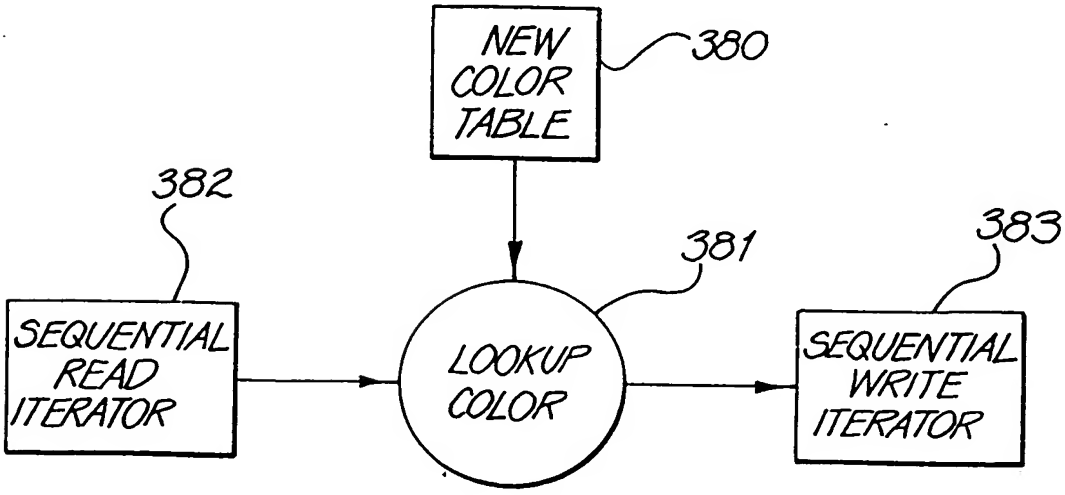

NEW COLOR TABLE
380
382
381
383
SEQUENTIAL READ ITERATOR
LOOKUP COLOR
SEQUENTIAL WRITE ITERATOR

Fig. 51

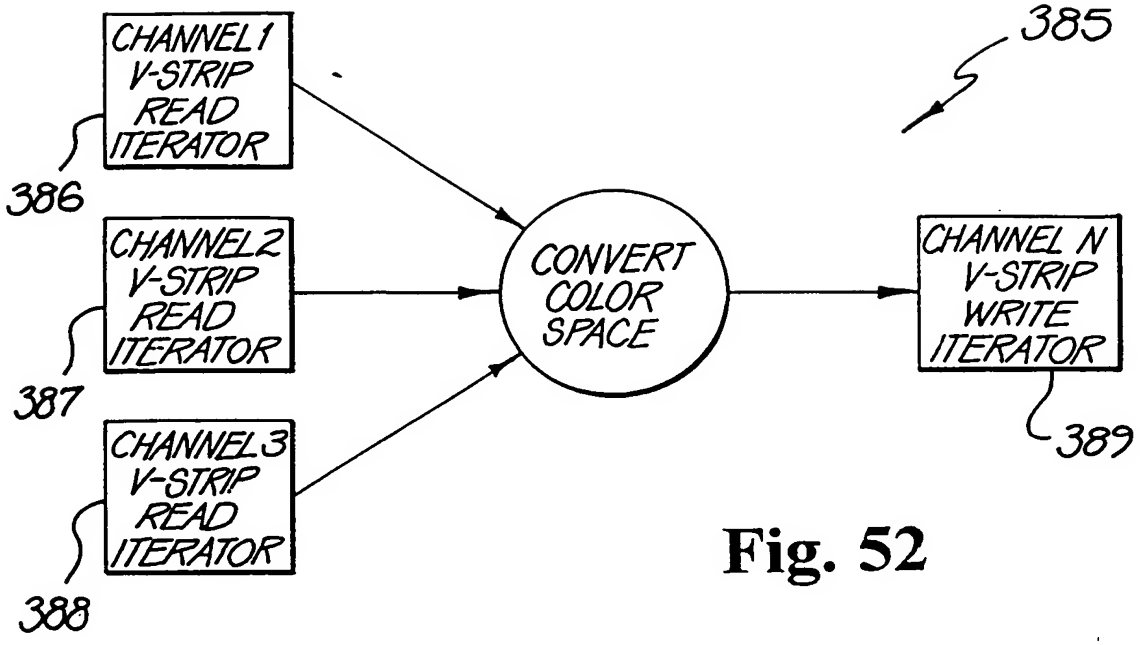

385
CHANNEL 1 V-STRIP READ ITERATOR
386
CHANNEL 2 V-STRIP READ ITERATOR
387
CHANNEL 3 V-STRIP READ ITERATOR
388
CONVERT COLOR SPACE
CHANNEL N V-STRIP WRITE ITERATOR
389

Fig. 52

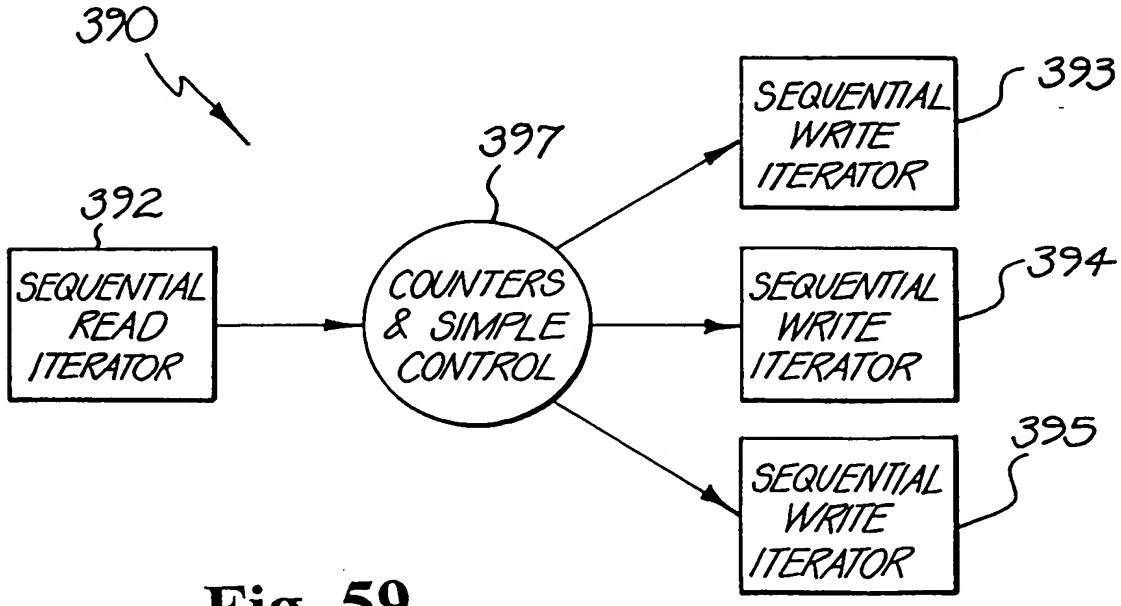

390
392
SEQUENTIAL READ ITERATOR
397
COUNTERS & SIMPLE CONTROL
SEQUENTIAL WRITE ITERATOR
393
SEQUENTIAL WRITE ITERATOR
394
SEQUENTIAL WRITE ITERATOR
395

Fig. 59

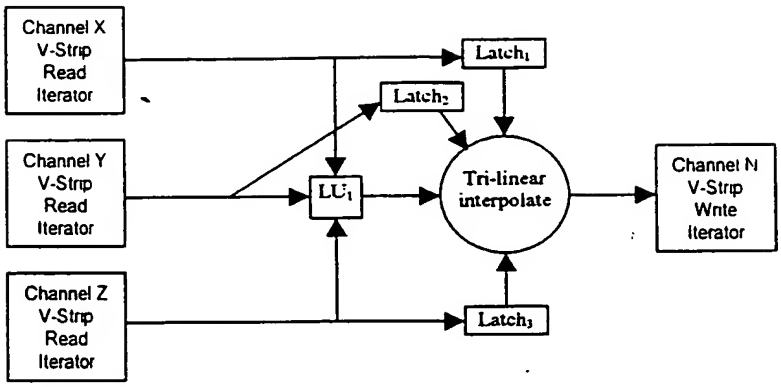

Channel X V-Strip Read Iterator
Channel Y V-Strip Read Iterator
Channel Z V-Strip Read Iterator
Latch1
Latch2
Latch3
LU1
Tri-linear interpolate
Channel N V-Strip Write Iterator

Fig. 53

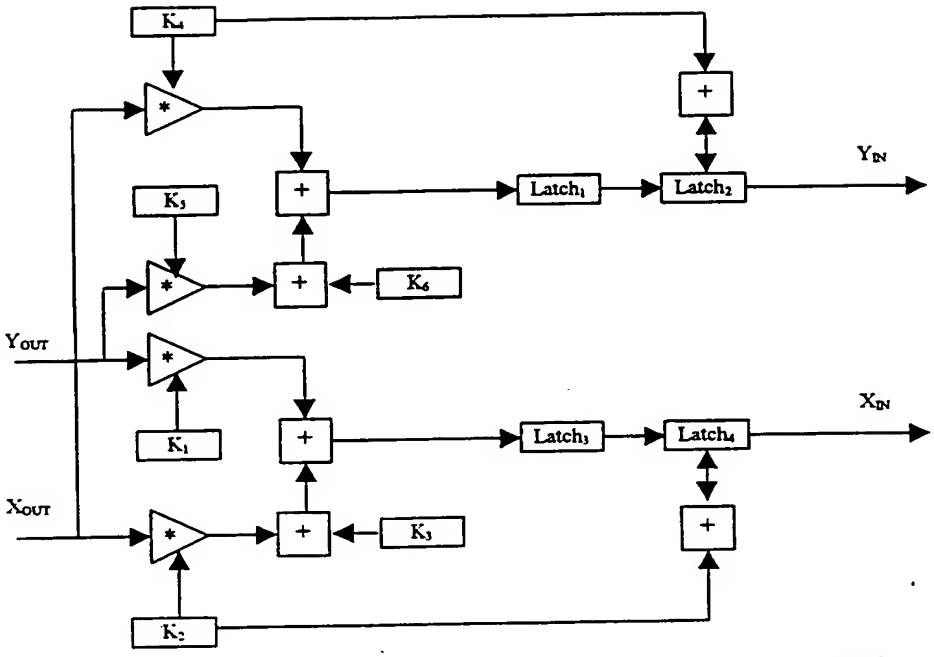

K4
K5
K6
K1
K3
K2
Latch1
Latch2
Latch3
Latch4
YOUT
XOUT
YIN
XIN

Fig. 54

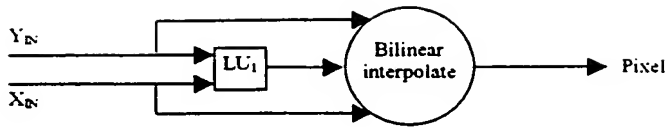

$Y_{IN}$
$X_{IN}$
$LU_1$
Bilinear interpolate
Pixel

Fig. 55

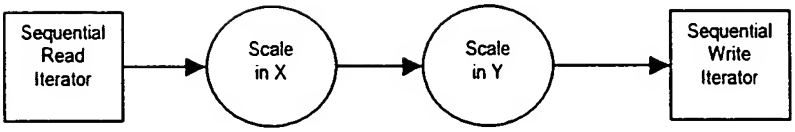

Sequential Read Iterator
Scale in X
Scale in Y
Sequential Write Iterator

Fig. 56

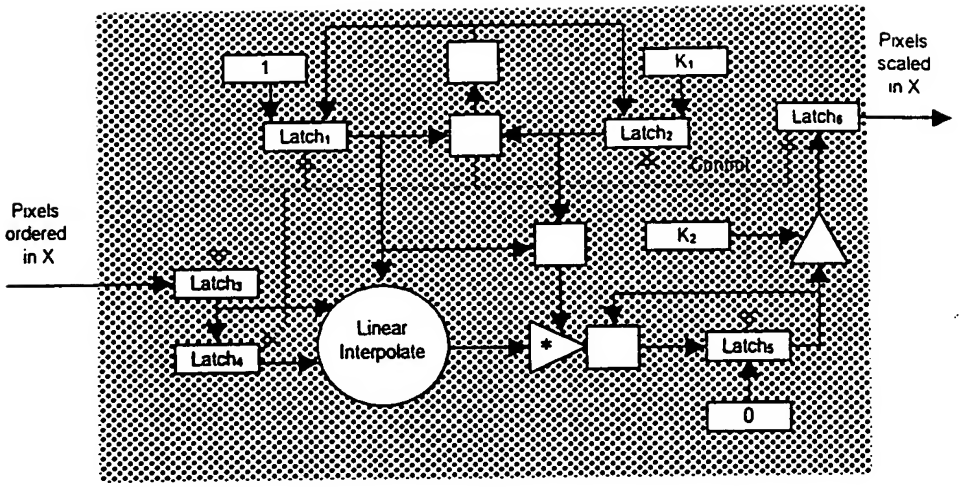

Pixels ordered in X
1
Latch1
K1
Latch2
Control
Latch6
Pixels scaled in X
K2
Latch3
Latch4
Linear Interpolate
*
Latch5
0

Fig. 57

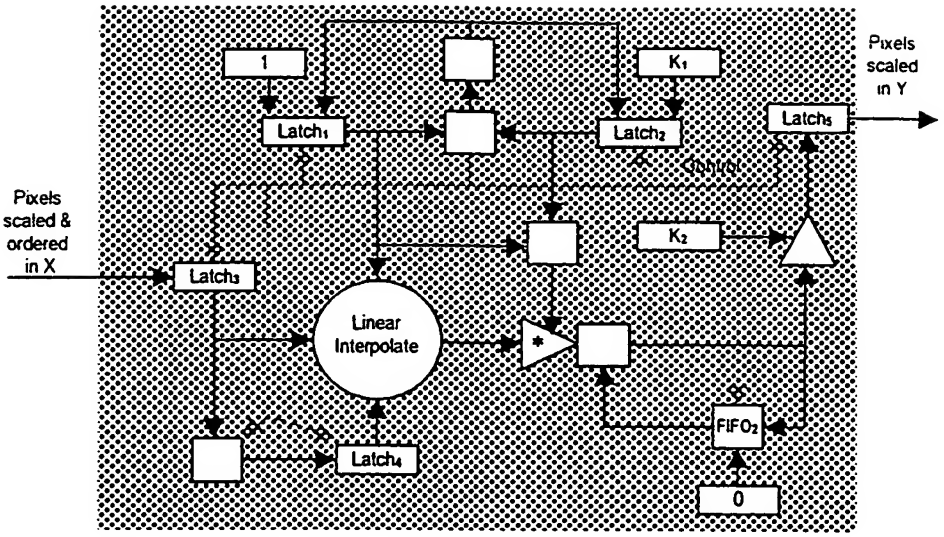

1
Latch1
K1
Latch2
Latch5
Pixels scaled in Y
Control
Pixels scaled & ordered in X
Latch3
K2
Linear Interpolate
*
FIFO2
Latch4
0

Fig. 58

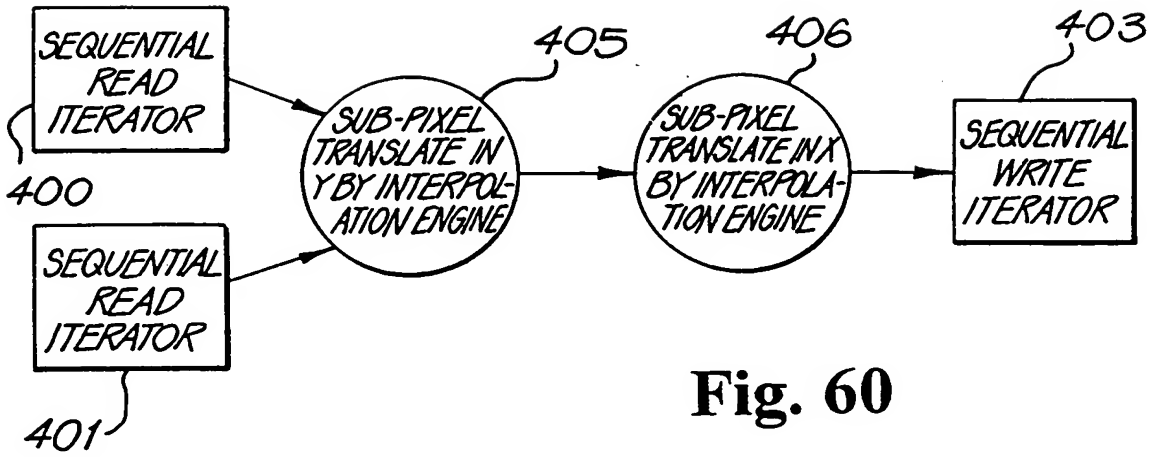

SEQUENTIAL READ ITERATOR
400
SEQUENTIAL READ ITERATOR
401
SUB-PIXEL TRANSLATE IN Y BY INTERPOLATION ENGINE
405
SUB-PIXEL TRANSLATE IN X BY INTERPOLATION ENGINE
406
SEQUENTIAL WRITE ITERATOR
403

Fig. 60

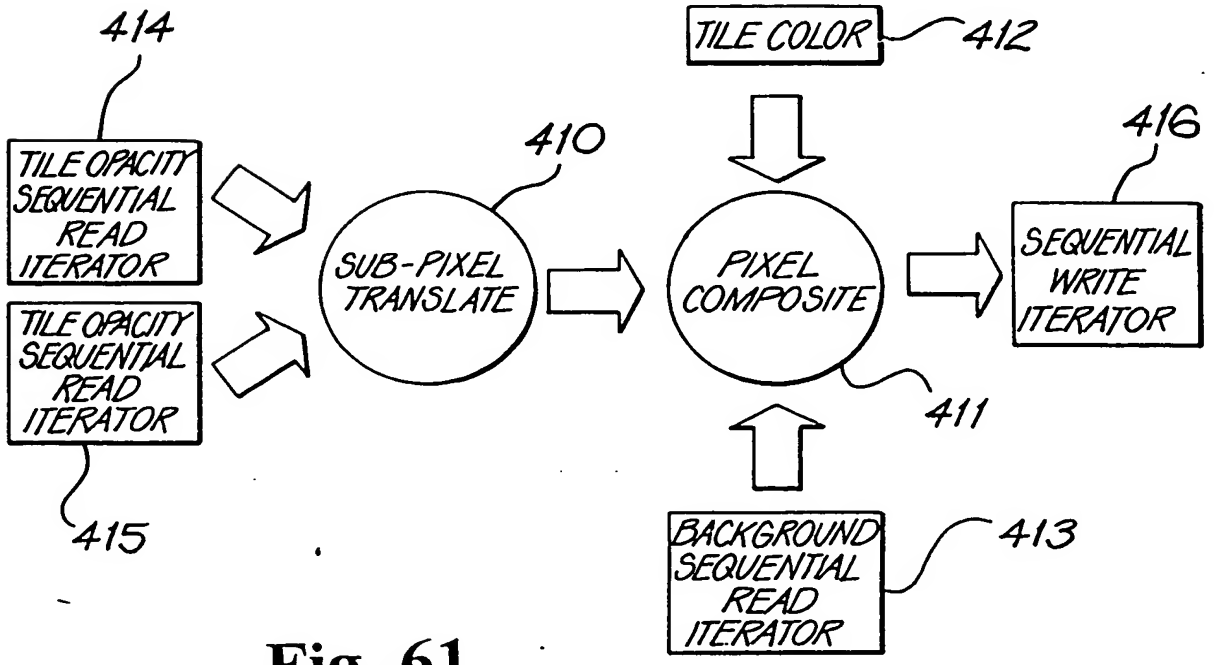

TILE OPACITY SEQUENTIAL READ ITERATOR
414
TILE OPACITY SEQUENTIAL READ ITERATOR
415
SUB-PIXEL TRANSLATE
410
TILE COLOR
412
PIXEL COMPOSITE
411
BACKGROUND SEQUENTIAL READ ITERATOR
413
SEQUENTIAL WRITE ITERATOR
416

Fig. 61

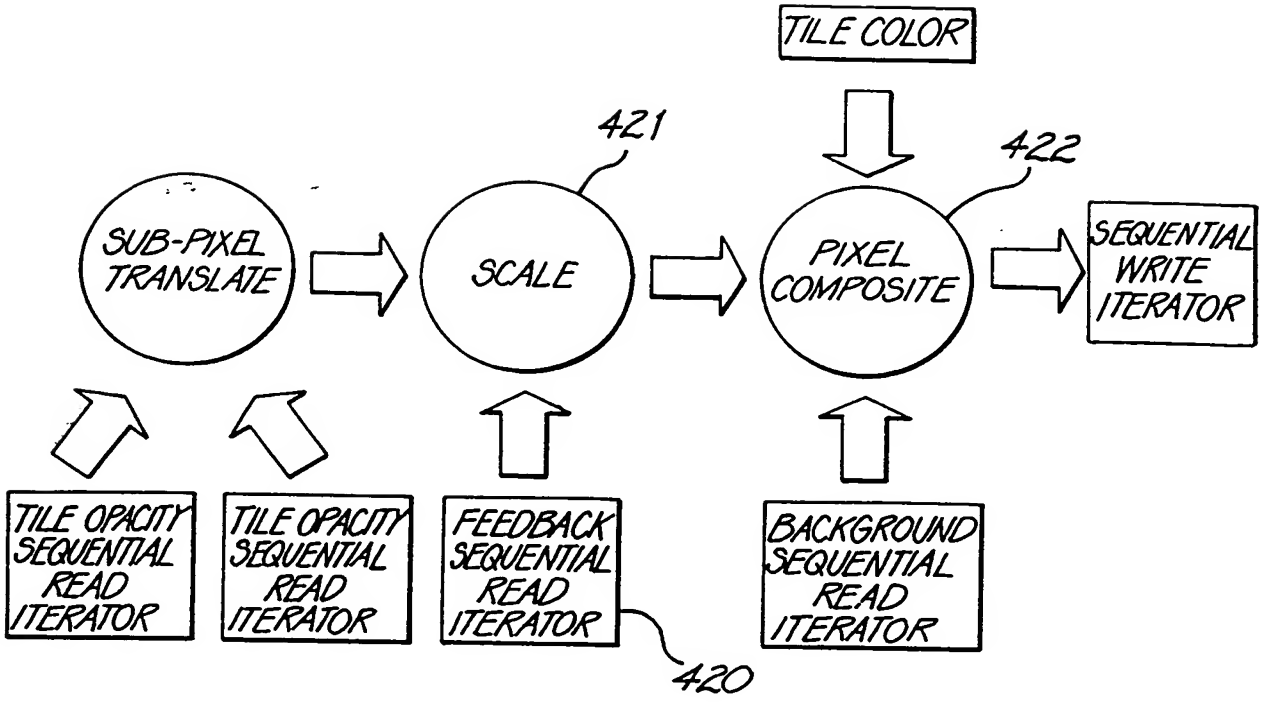

TILE COLOR
421
422
SUB-PIXEL TRANSLATE
SCALE
PIXEL COMPOSITE
SEQUENTIAL WRITE ITERATOR
TILE OPACITY SEQUENTIAL READ ITERATOR
TILE OPACITY SEQUENTIAL READ ITERATOR
FEEDBACK SEQUENTIAL READ ITERATOR
BACKGROUND SEQUENTIAL READ ITERATOR
420

Fig. 62

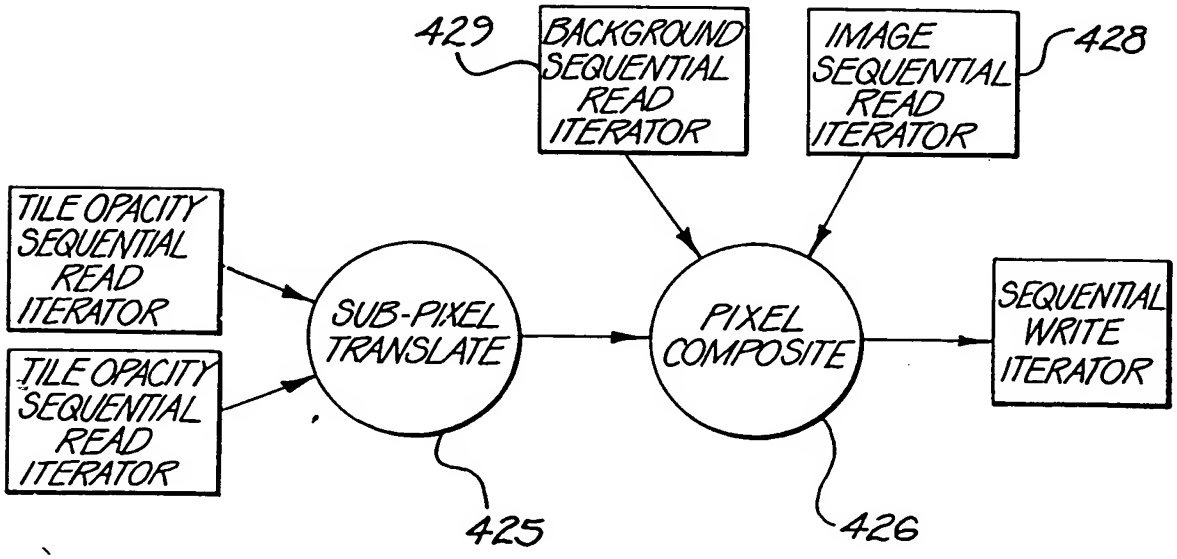

429
BACKGROUND SEQUENTIAL READ ITERATOR
IMAGE SEQUENTIAL READ ITERATOR
428
TILE OPACITY SEQUENTIAL READ ITERATOR
TILE OPACITY SEQUENTIAL READ ITERATOR
SUB-PIXEL TRANSLATE
PIXEL COMPOSITE
SEQUENTIAL WRITE ITERATOR
425
426

Fig. 63

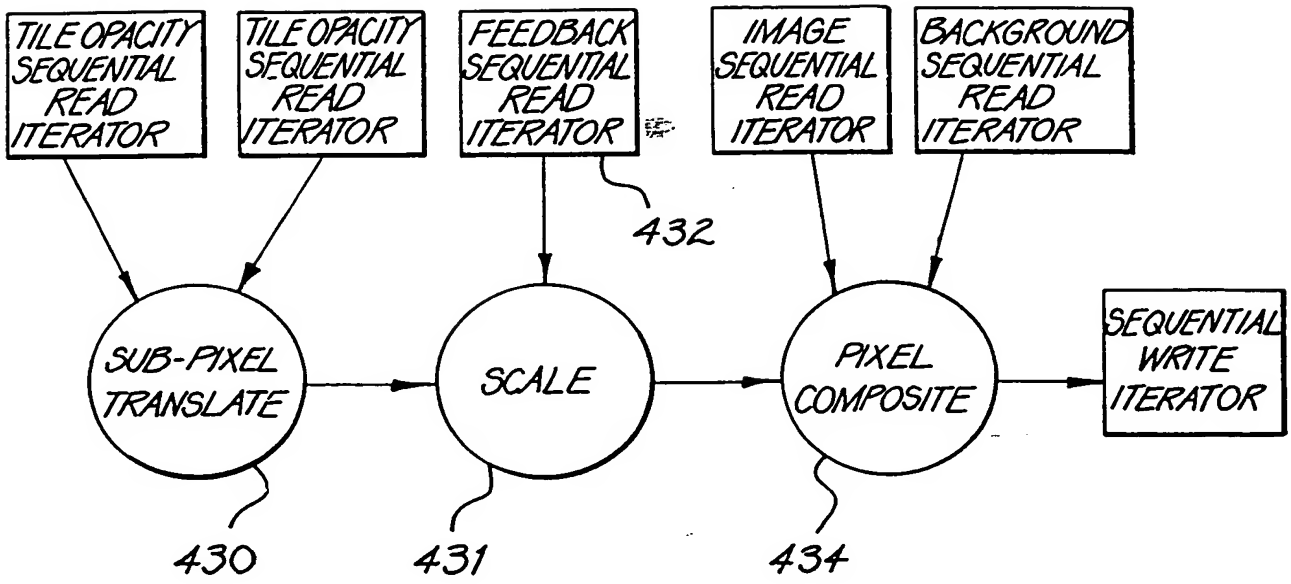

TILE OPACITY SEQUENTIAL READ ITERATOR
TILE OPACITY SEQUENTIAL READ ITERATOR
FEEDBACK SEQUENTIAL READ ITERATOR
IMAGE SEQUENTIAL READ ITERATOR
BACKGROUND SEQUENTIAL READ ITERATOR
432
SUB-PIXEL TRANSLATE
SCALE
PIXEL COMPOSITE
SEQUENTIAL WRITE ITERATOR
430
431
434

**Fig. 64**

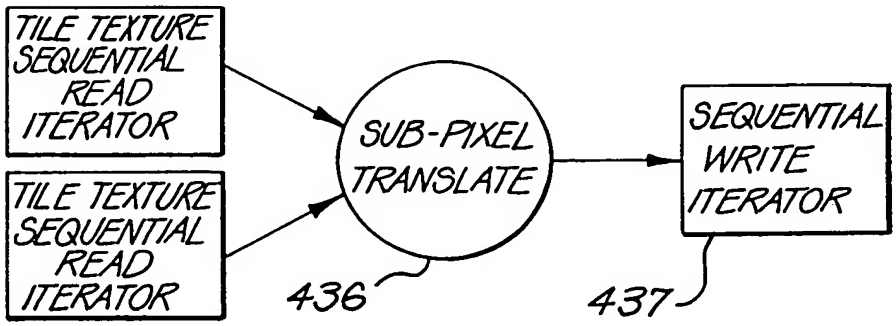

TILE TEXTURE SEQUENTIAL READ ITERATOR
TILE TEXTURE SEQUENTIAL READ ITERATOR
SUB-PIXEL TRANSLATE
SEQUENTIAL WRITE ITERATOR
436
437

**Fig. 65**

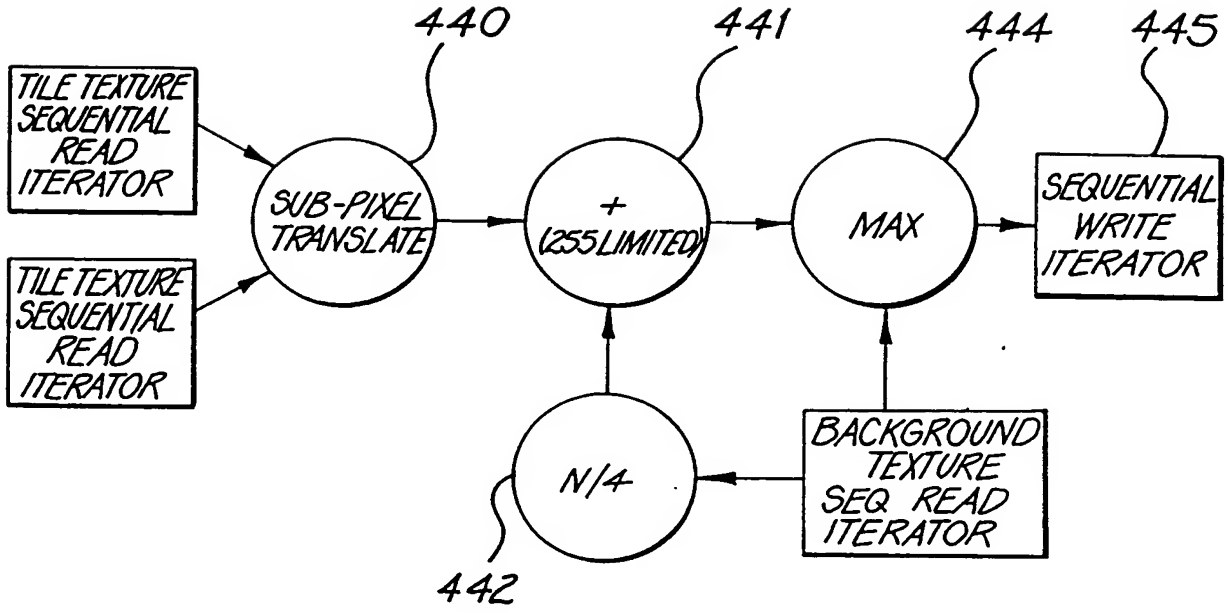

440
441
444
445
TILE TEXTURE SEQUENTIAL READ ITERATOR
TILE TEXTURE SEQUENTIAL READ ITERATOR
SUB-PIXEL TRANSLATE
+ (255 LIMITED)
MAX
SEQUENTIAL WRITE ITERATOR
N/4
BACKGROUND TEXTURE SEQ READ ITERATOR
442

Fig. 66

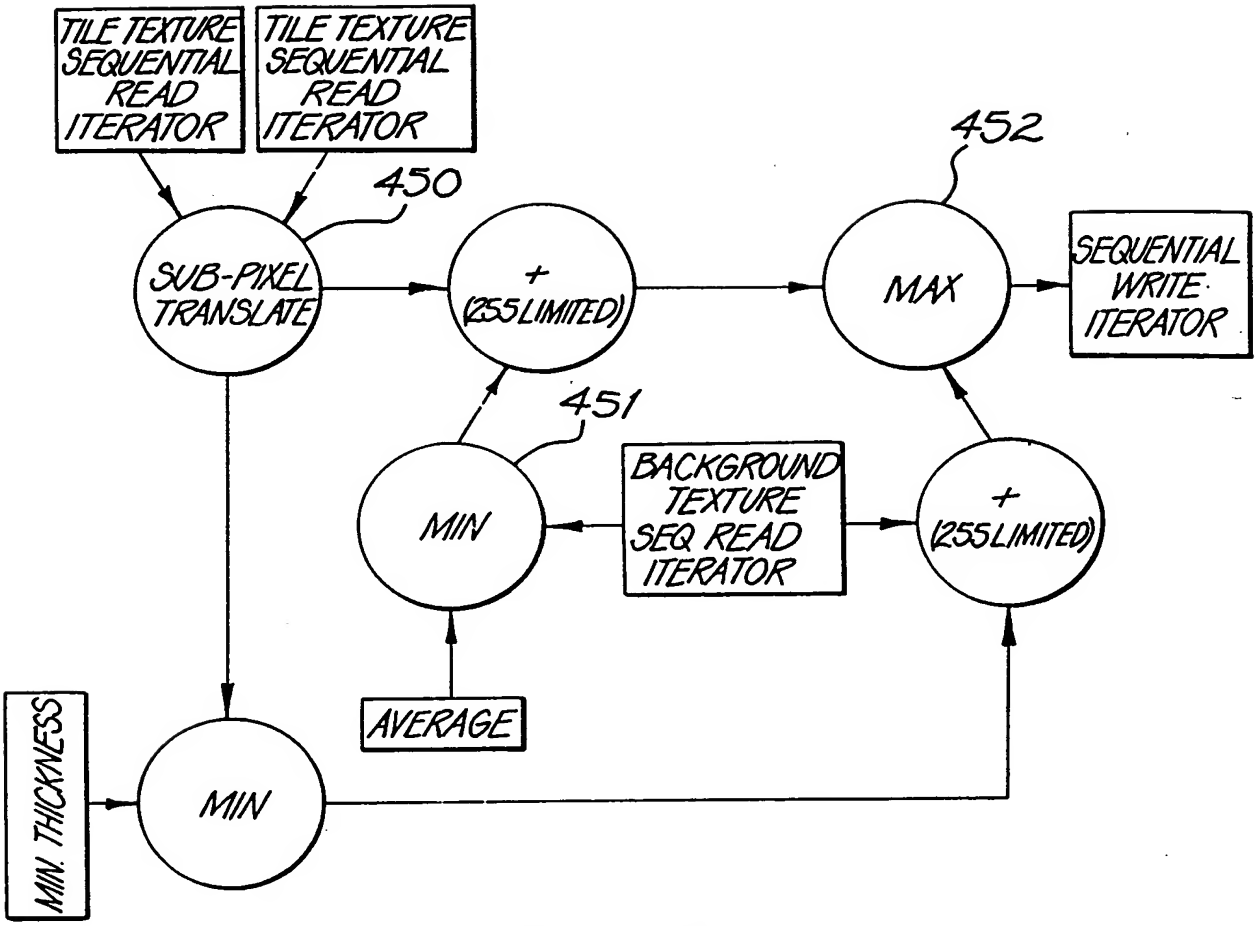

TILE TEXTURE SEQUENTIAL READ ITERATOR
TILE TEXTURE SEQUENTIAL READ ITERATOR
450
452
SUB-PIXEL TRANSLATE
+ (255 LIMITED)
MAX
SEQUENTIAL WRITE ITERATOR
451
MIN
BACKGROUND TEXTURE SEQ READ ITERATOR
+ (255 LIMITED)
AVERAGE
MIN. THICKNESS
MIN

Fig. 67

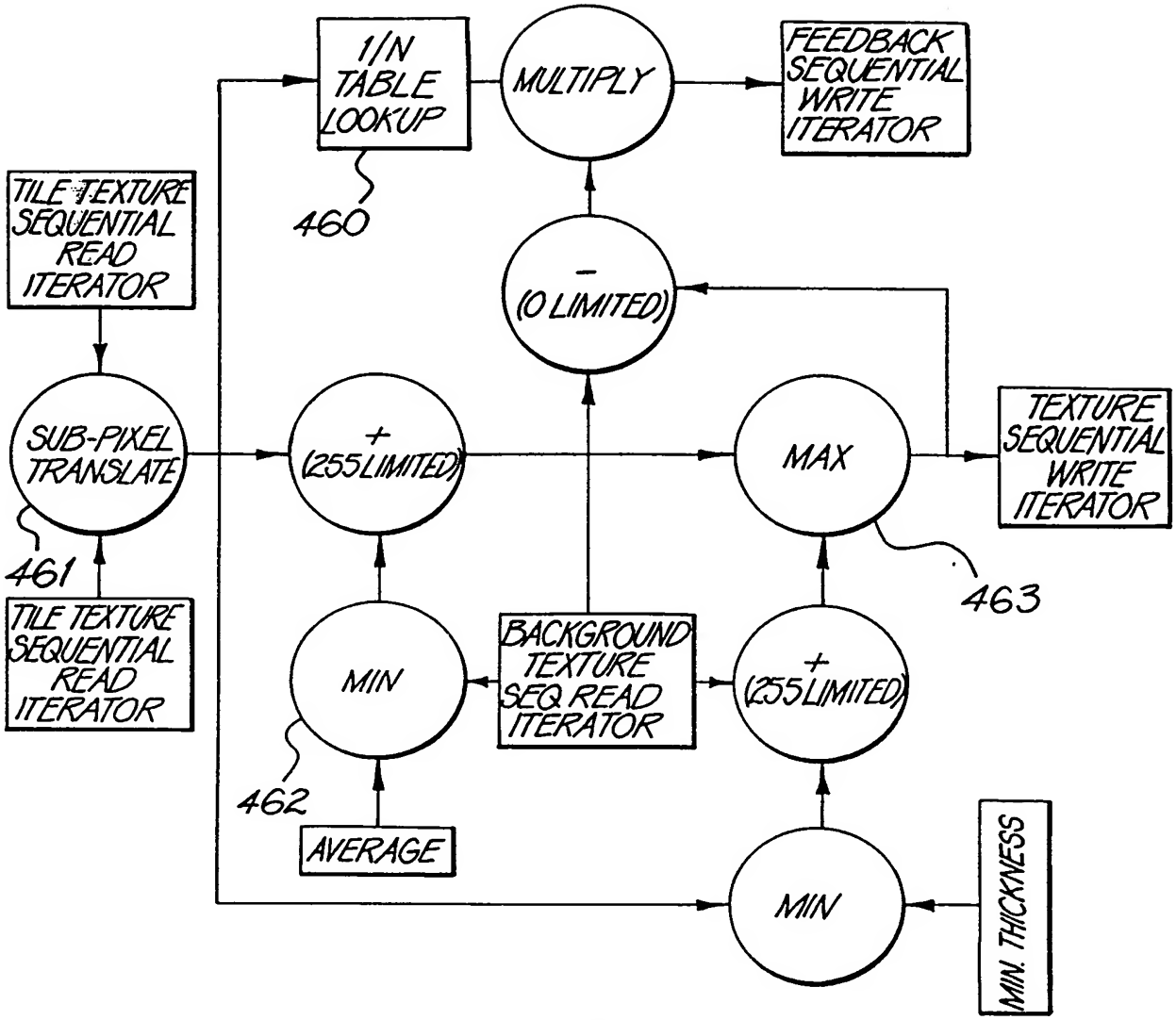

1/N TABLE LOOKUP
MULTIPLY
FEEDBACK SEQUENTIAL WRITE ITERATOR
460
TILE TEXTURE SEQUENTIAL READ ITERATOR
– (0 LIMITED)
SUB-PIXEL TRANSLATE
+ (255 LIMITED)
MAX
TEXTURE SEQUENTIAL WRITE ITERATOR
461
463
TILE TEXTURE SEQUENTIAL READ ITERATOR
MIN
BACKGROUND TEXTURE SEQ READ ITERATOR
+ (255 LIMITED)
462
AVERAGE
MIN
MIN. THICKNESS

Fig. 68

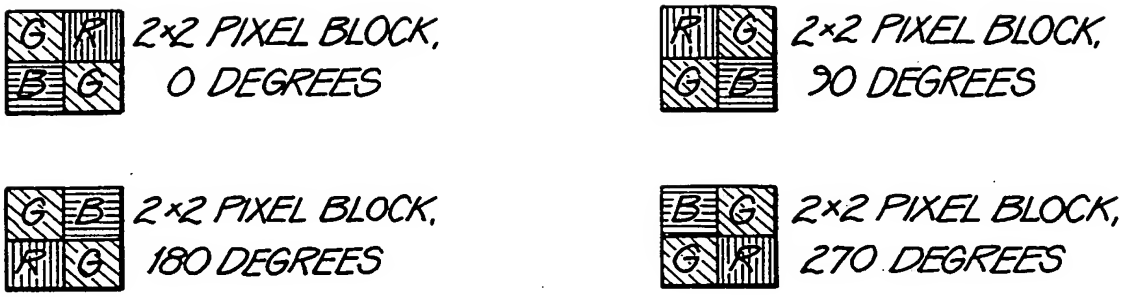

G R
B G
2×2 PIXEL BLOCK, 0 DEGREES
R G
G B
2×2 PIXEL BLOCK, 90 DEGREES
G B
R G
2×2 PIXEL BLOCK, 180 DEGREES
B G
G R
2×2 PIXEL BLOCK, 270 DEGREES

Fig. 69

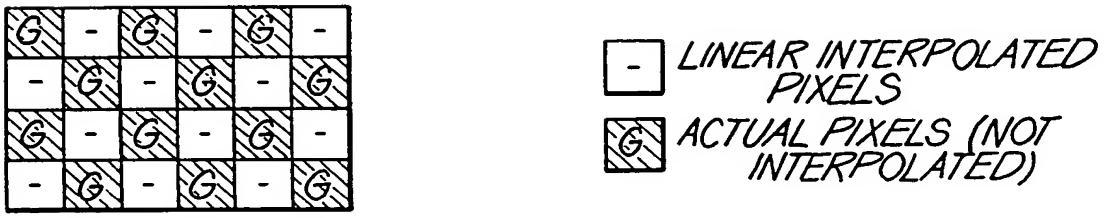

G - G - G -
- G - G - G
G - G - G -
- G - G - G
- LINEAR INTERPOLATED PIXELS
G ACTUAL PIXELS (NOT INTERPOLATED)

Fig. 70

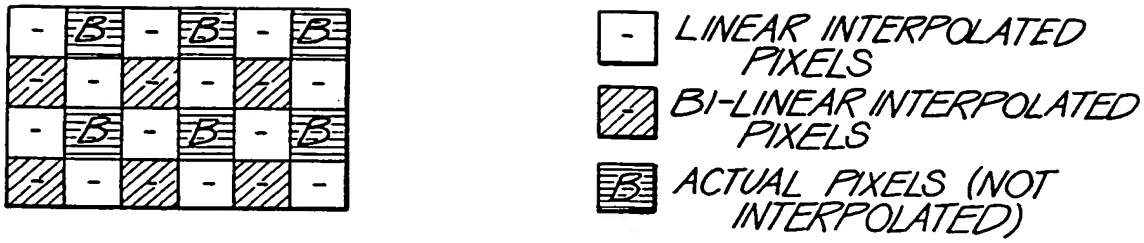

B
LINEAR INTERPOLATED PIXELS
BI-LINEAR INTERPOLATED PIXELS
ACTUAL PIXELS (NOT INTERPOLATED)

Fig. 71

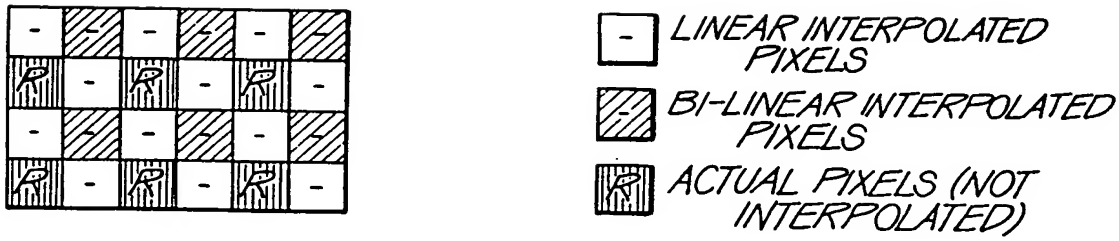

R
LINEAR INTERPOLATED PIXELS
BI-LINEAR INTERPOLATED PIXELS
ACTUAL PIXELS (NOT INTERPOLATED)

Fig. 72

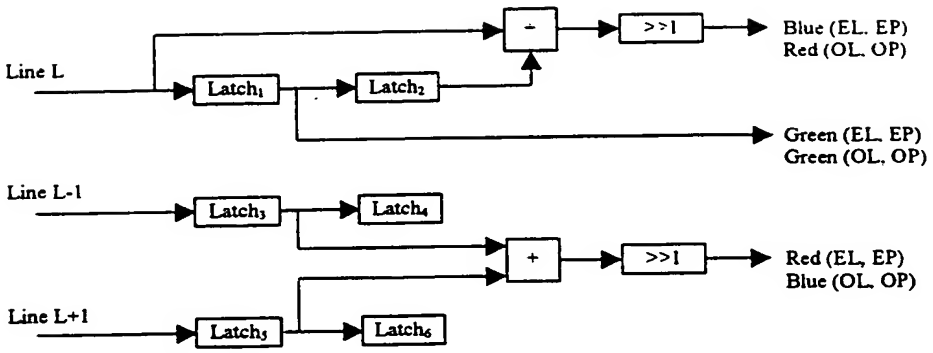

Line L
Latch1
Latch2
+
>>1
Blue (EL, EP)
Red (OL, OP)
Green (EL, EP)
Green (OL, OP)
Line L-1
Latch3
Latch4
Line L+1
Latch5
Latch6
+
>>1
Red (EL, EP)
Blue (OL, OP)

Fig. 73

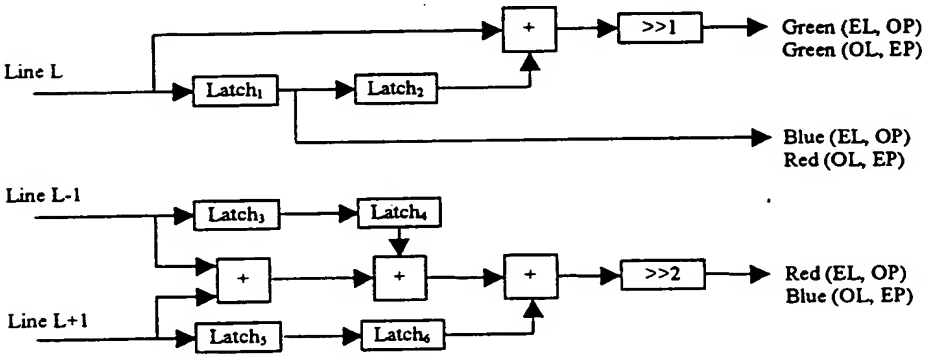

Line L
Latch1
Latch2
+
>>1
Green (EL, OP)
Green (OL, EP)
Blue (EL, OP)
Red (OL, EP)
Line L-1
Latch3
Latch4
Line L+1
Latch5
Latch6
+
+
+
>>2
Red (EL, OP)
Blue (OL, EP)

Fig. 74

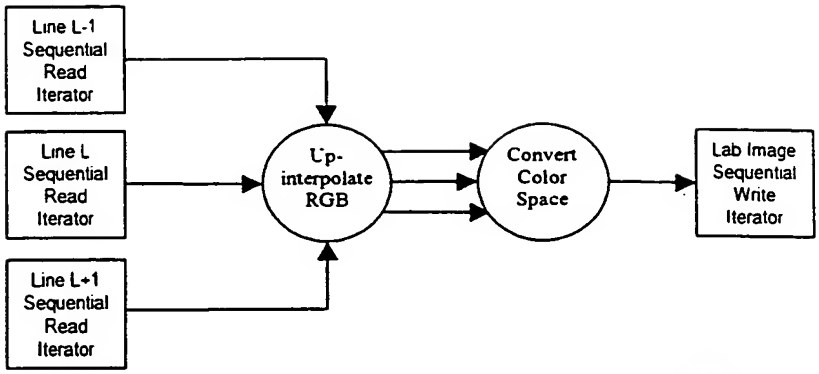

Line L-1 Sequential Read Iterator
Line L Sequential Read Iterator
Line L+1 Sequential Read Iterator
Up-interpolate RGB
Convert Color Space
Lab Image Sequential Write Iterator

Fig. 75

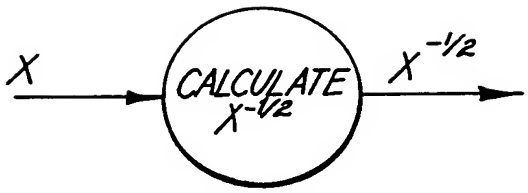



Fig. 76

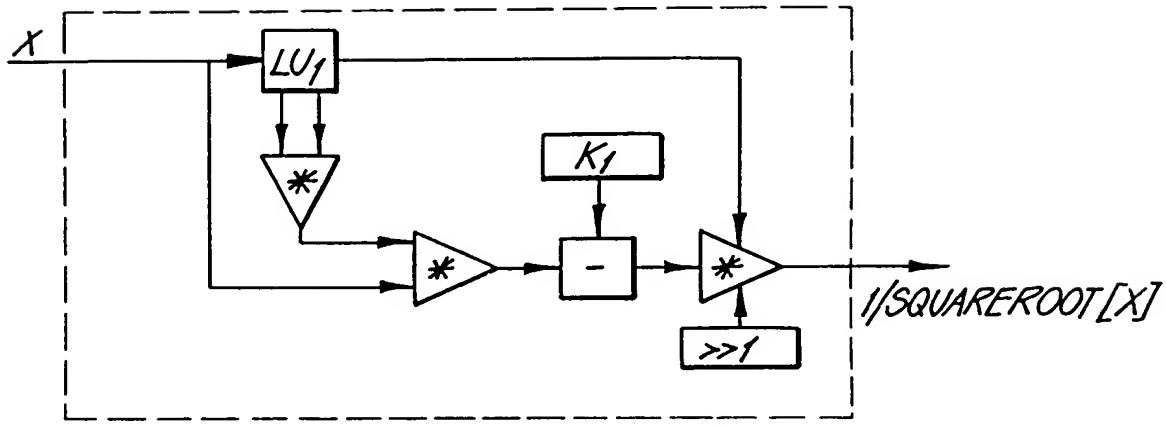



Fig. 77

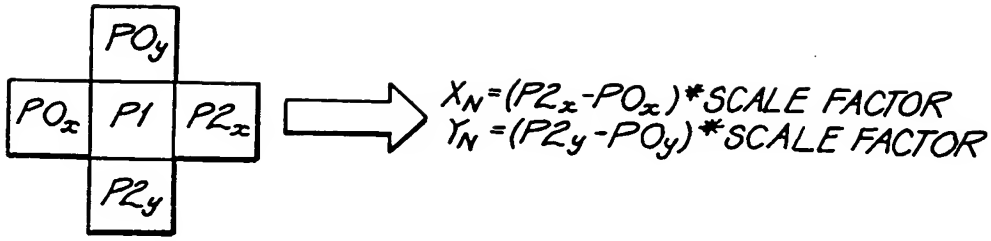

PO_y
PO_x
P1
P2_x
P2_y
X_N=(P2_x-PO_x)*SCALE FACTOR
Y_N=(P2_y-PO_y)*SCALE FACTOR

**Fig. 78**

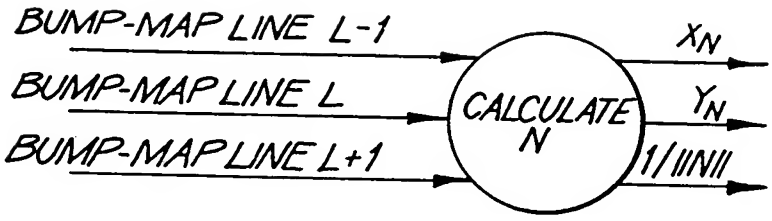

BUMP-MAP LINE L-1
BUMP-MAP LINE L
BUMP-MAP LINE L+1
CALCULATE N
X_N
Y_N
1/||N||

**Fig. 79**

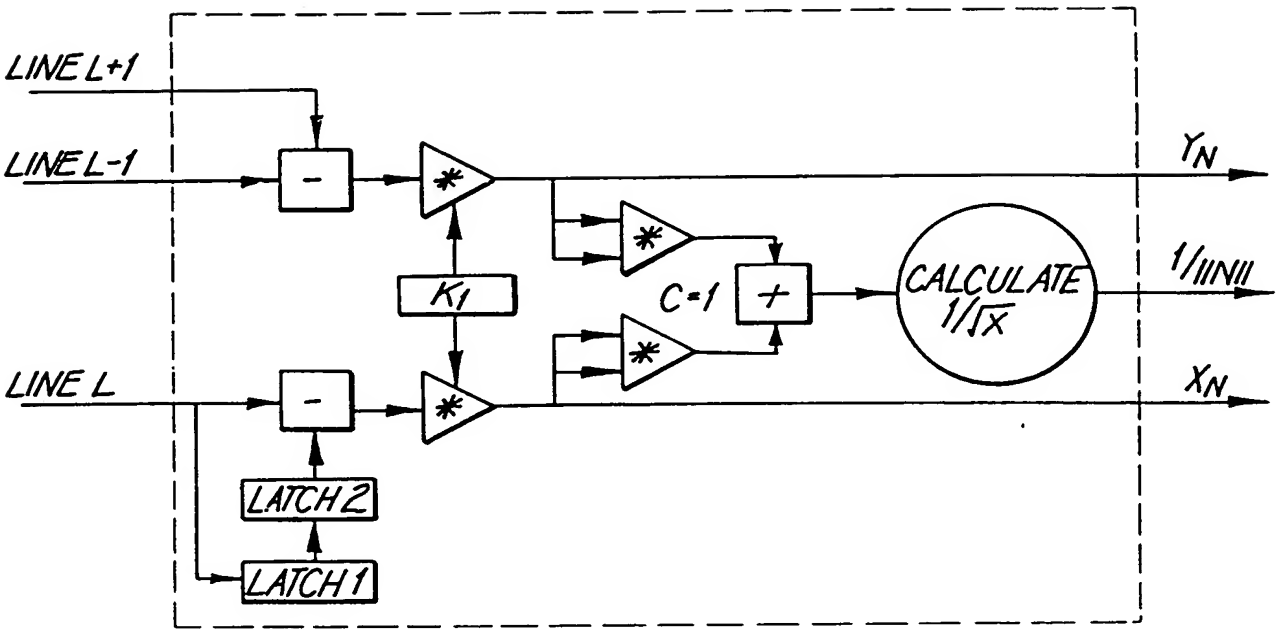

LINE L+1
LINE L-1
LINE L
K1
C=1
CALCULATE
1/√X
YN
1/IINII
XN
LATCH 2
LATCH 1

**Fig. 80**

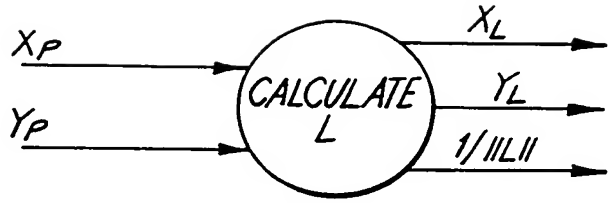



**Fig. 81**

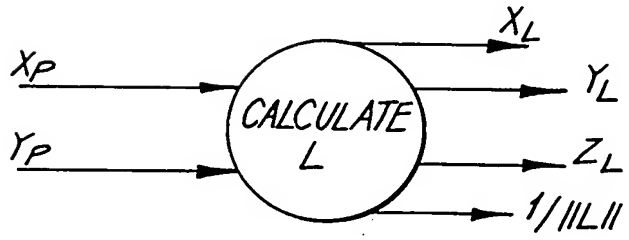



**Fig. 82**

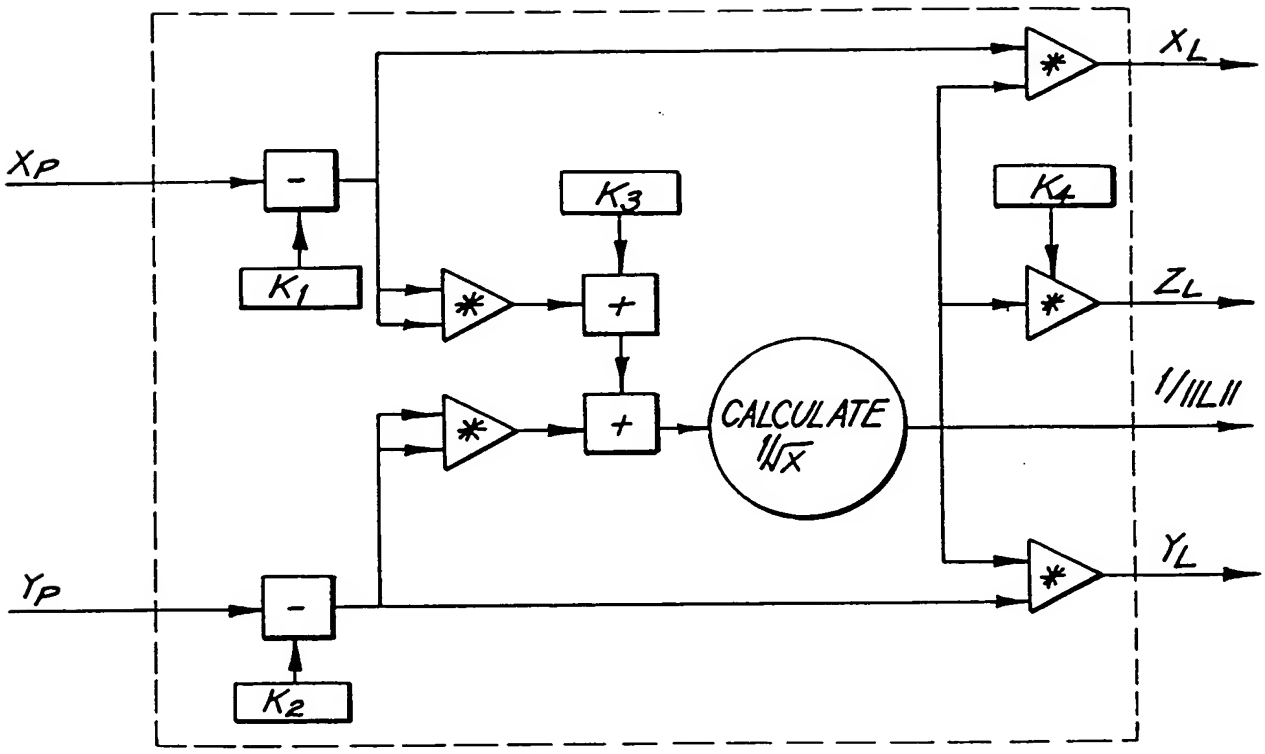



**Fig. 83**

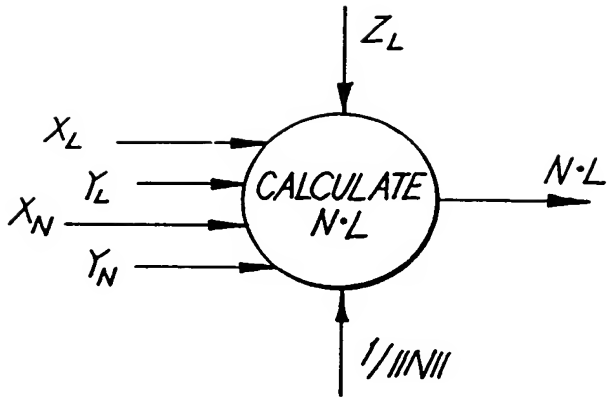



Fig. 84

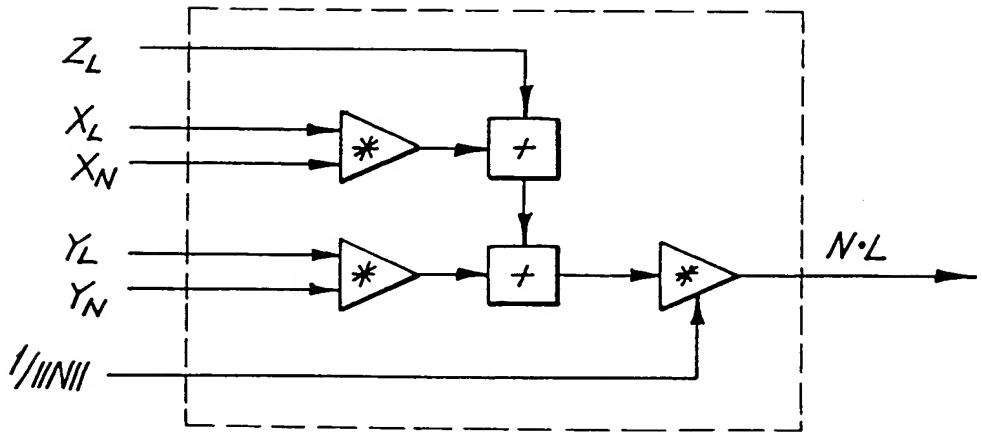



Fig. 85

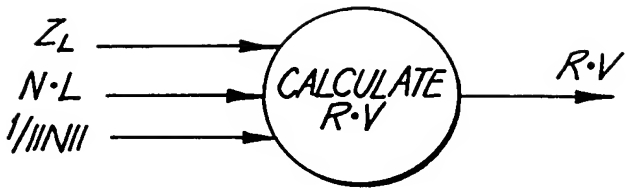



Fig. 86

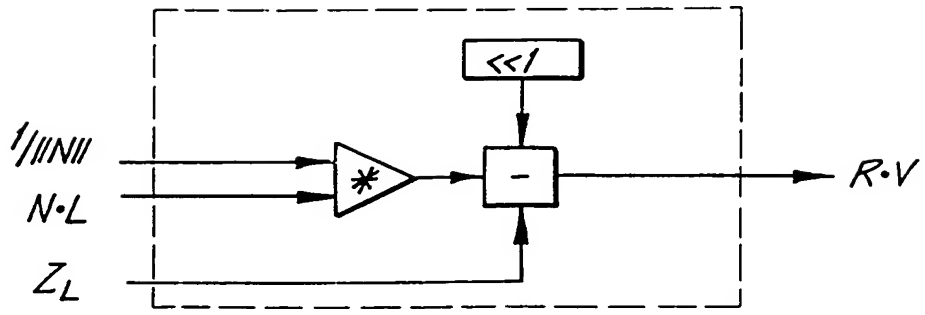



Fig. 87

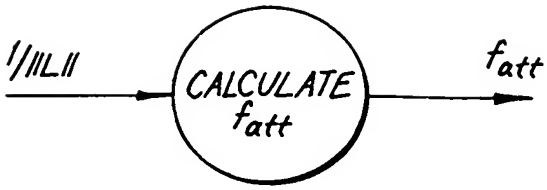



Fig. 88

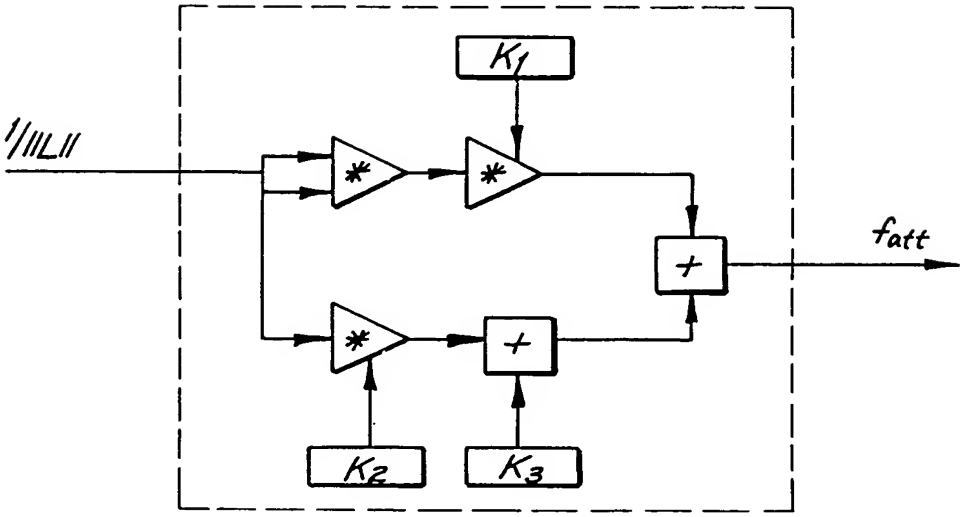



Fig. 89

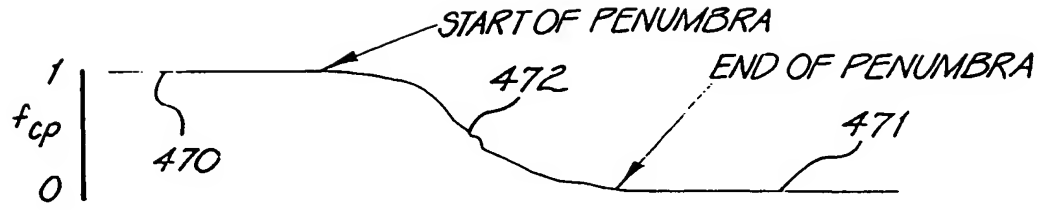



Fig. 90

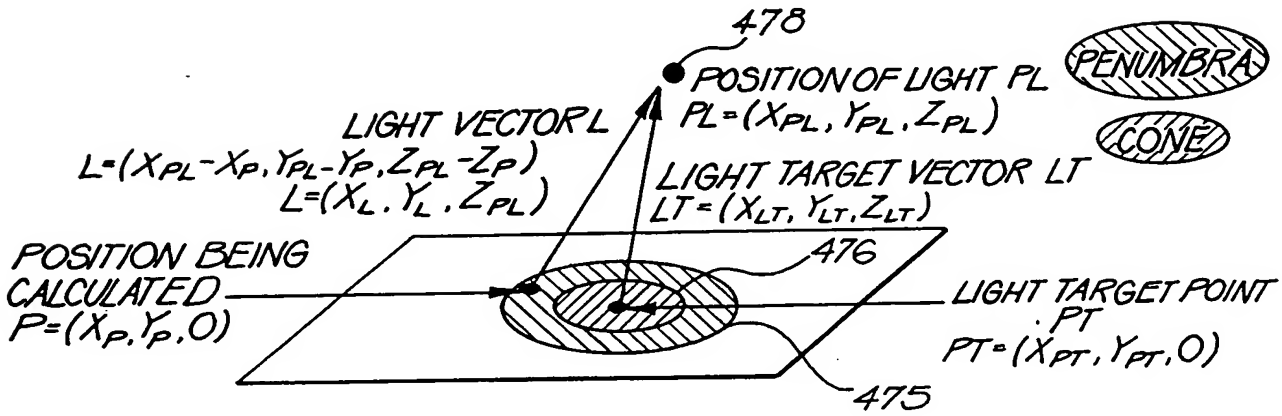



Fig. 91

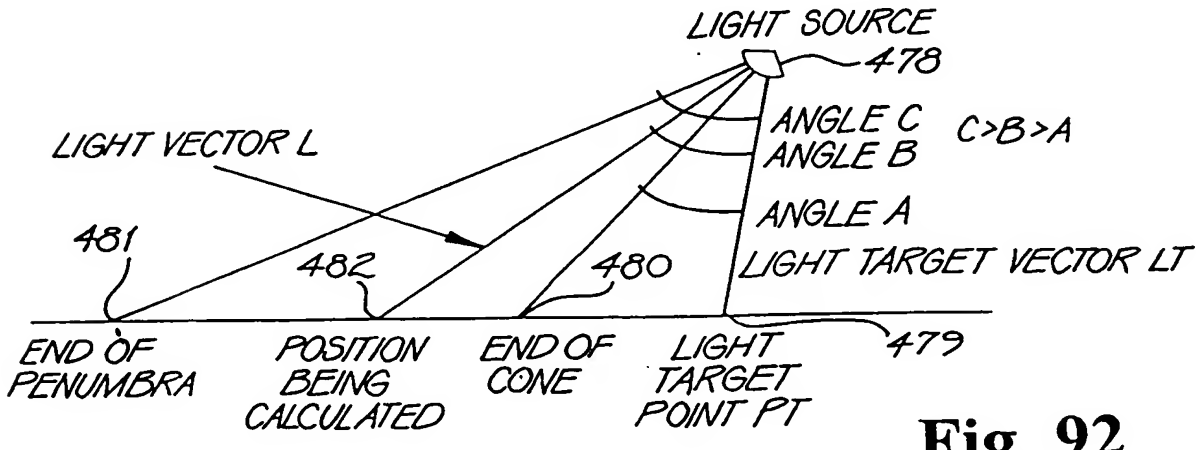



Fig. 92

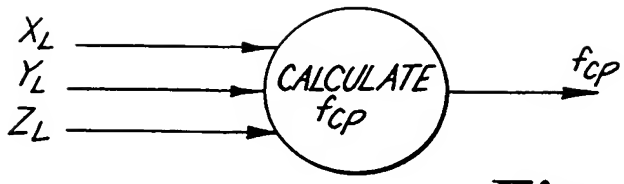

$X_L$
$Y_L$
$Z_L$
CALCULATE
$f_{CP}$
$f_{CP}$

Fig. 93

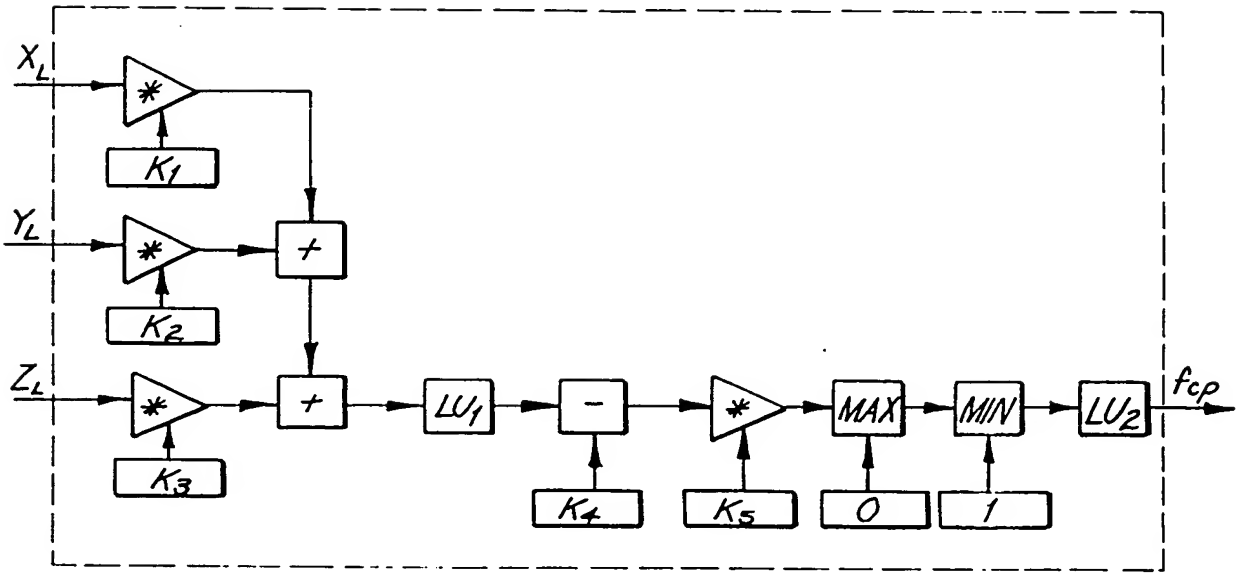

$X_L$
$Y_L$
$Z_L$
$K_1$
$K_2$
$K_3$
$K_4$
$K_5$
LU1
LU2
MAX
MIN
0
1
$f_{CP}$

Fig. 94

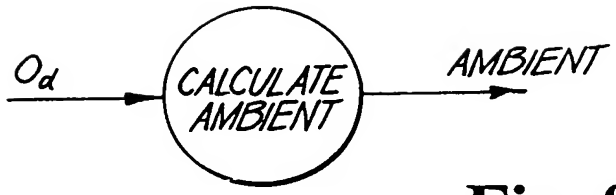



Fig. 95

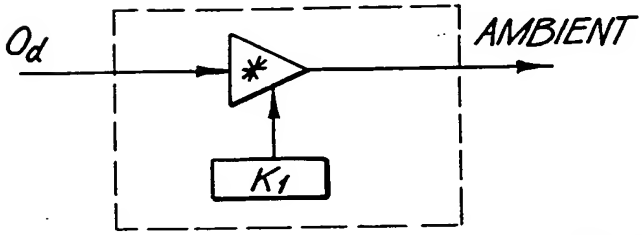



Fig. 96

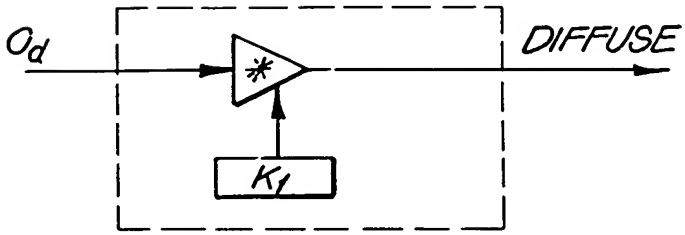



Fig. 97

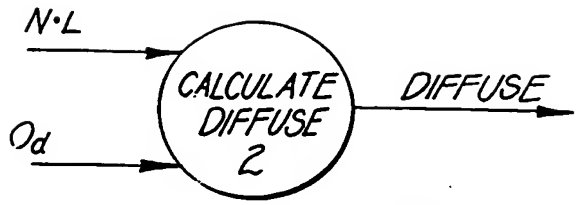



Fig. 98

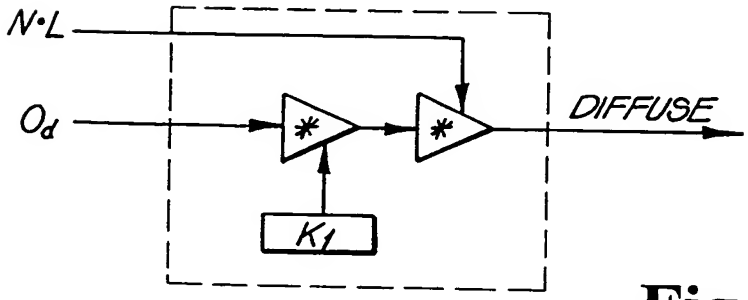



Fig. 99

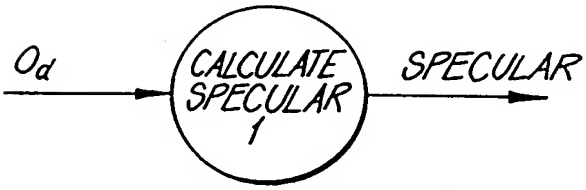



Fig. 100

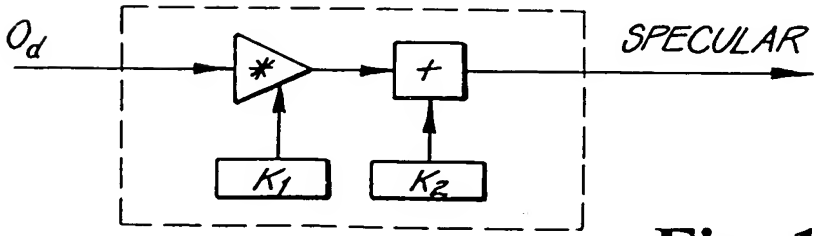



Fig. 101

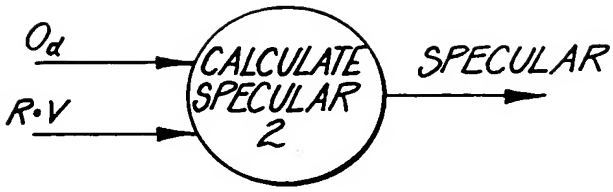



**Fig. 102**

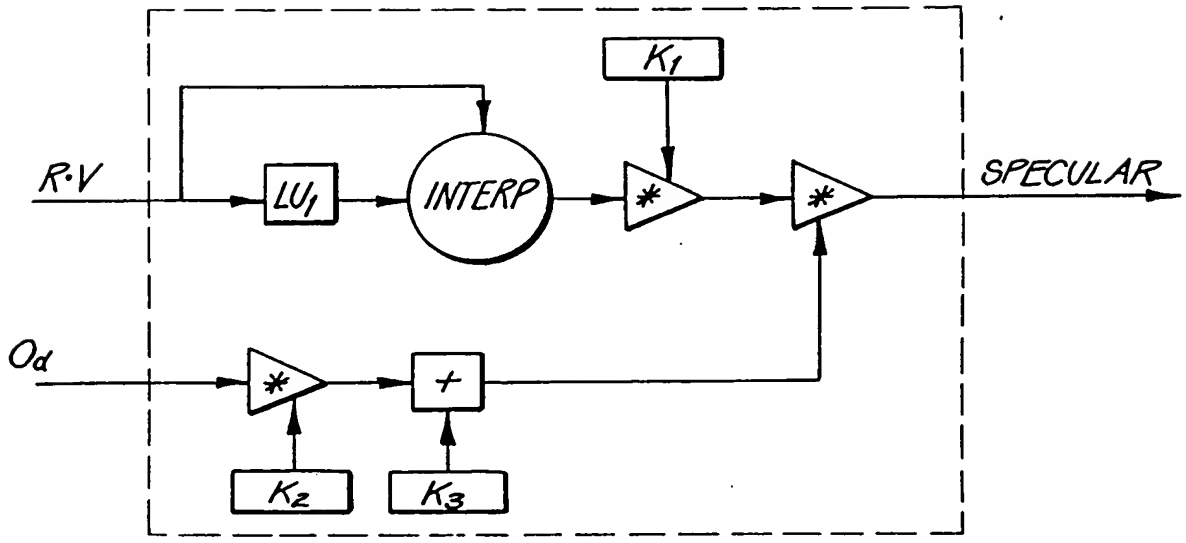



**Fig. 103**

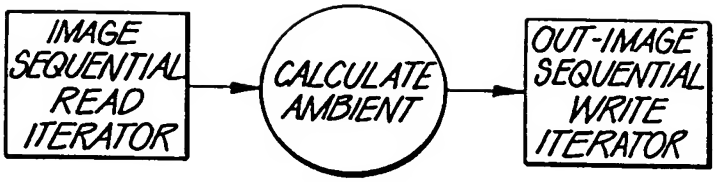



Fig. 104

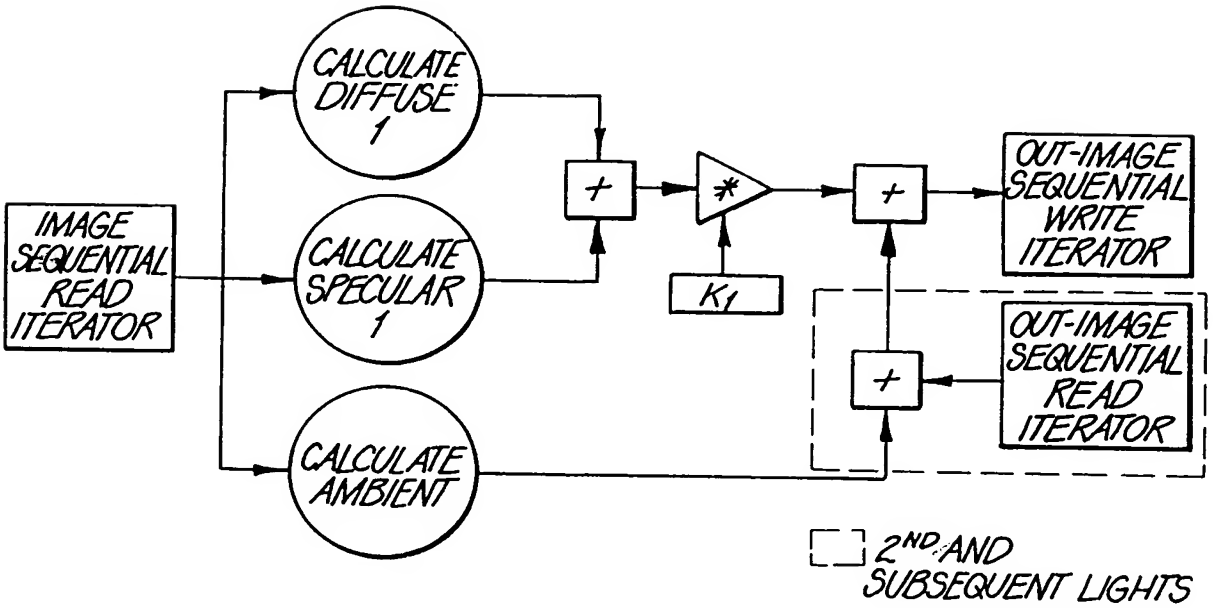



Fig. 105

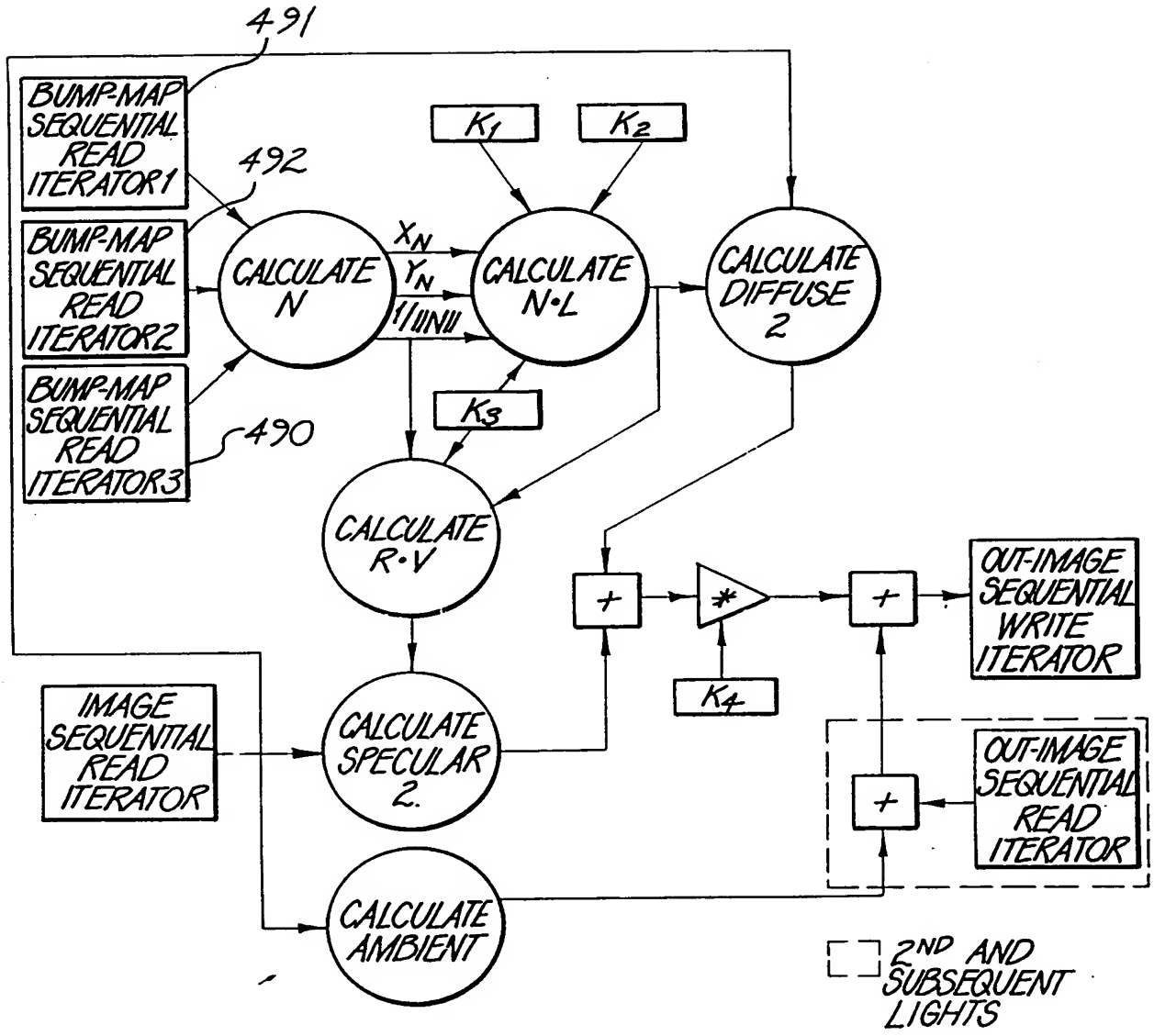

491
BUMP-MAP SEQUENTIAL READ ITERATOR1
492
BUMP-MAP SEQUENTIAL READ ITERATOR2
BUMP-MAP SEQUENTIAL READ ITERATOR3
490
K1
K2
CALCULATE N
XN
YN
1/||N||
CALCULATE N·L
CALCULATE DIFFUSE 2
K3
CALCULATE R·V
+
*
+
OUT-IMAGE SEQUENTIAL WRITE ITERATOR
K4
IMAGE SEQUENTIAL READ ITERATOR
CALCULATE SPECULAR 2.
+
OUT-IMAGE SEQUENTIAL READ ITERATOR
CALCULATE AMBIENT
2ND AND SUBSEQUENT LIGHTS

Fig. 106

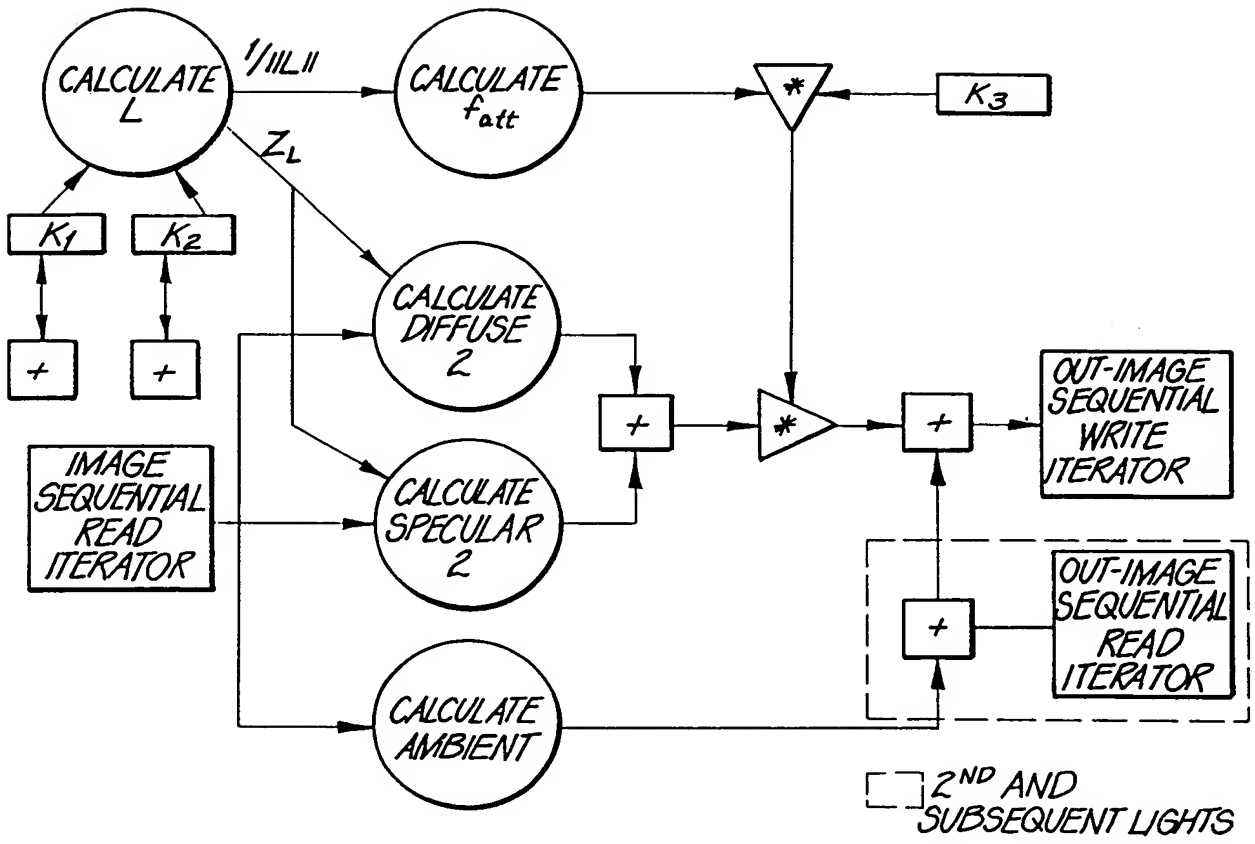

CALCULATE L
1/||L||
CALCULATE $f_{att}$
K3
$Z_L$
K1
K2
CALCULATE DIFFUSE 2
OUT-IMAGE SEQUENTIAL WRITE ITERATOR
IMAGE SEQUENTIAL READ ITERATOR
CALCULATE SPECULAR 2
OUT-IMAGE SEQUENTIAL READ ITERATOR
CALCULATE AMBIENT
2ND AND SUBSEQUENT LIGHTS

**Fig. 107**

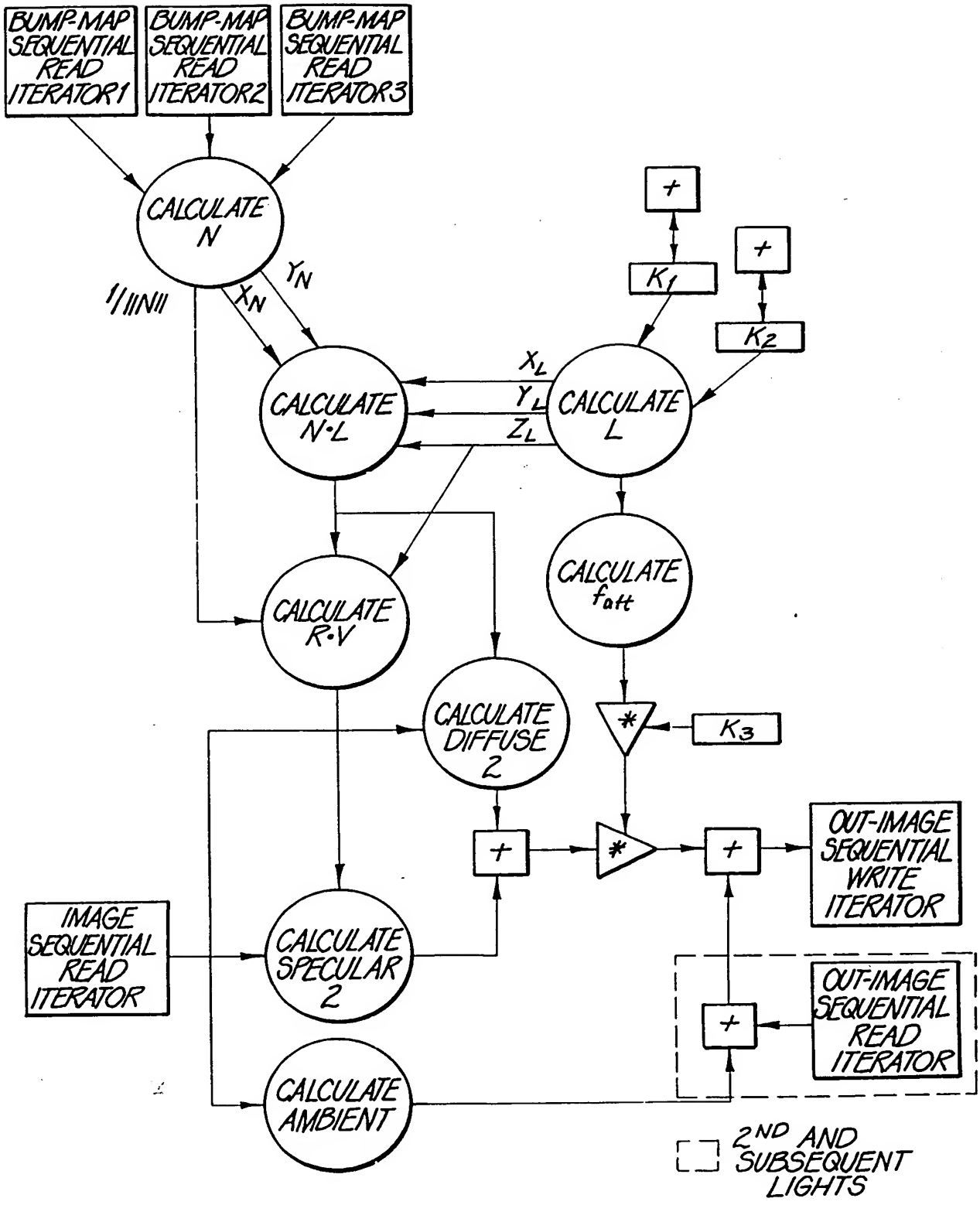

BUMP-MAP SEQUENTIAL READ ITERATOR1
BUMP-MAP SEQUENTIAL READ ITERATOR2
BUMP-MAP SEQUENTIAL READ ITERATOR3
CALCULATE N
1/||N||
$Y_N$
$X_N$
+
$K_1$
+
$K_2$
$X_L$
$Y_L$
$Z_L$
CALCULATE N·L
CALCULATE L
CALCULATE R·V
CALCULATE $f_{att}$
CALCULATE DIFFUSE 2
*
$K_3$
+
*
+
OUT-IMAGE SEQUENTIAL WRITE ITERATOR
IMAGE SEQUENTIAL READ ITERATOR
CALCULATE SPECULAR 2
+
OUT-IMAGE SEQUENTIAL READ ITERATOR
CALCULATE AMBIENT
2ND AND SUBSEQUENT LIGHTS

Fig. 108

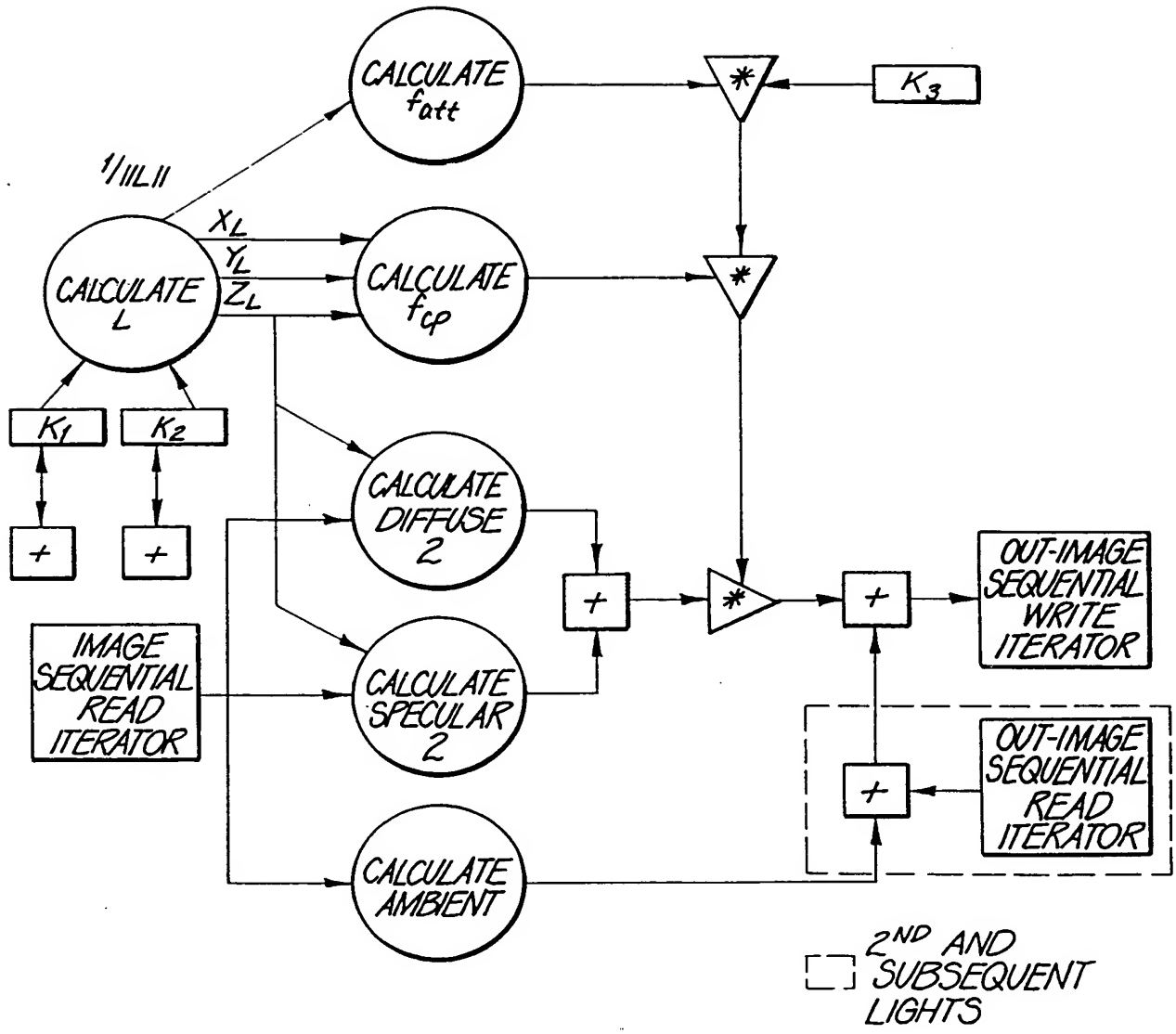

CALCULATE $f_{att}$
$K_3$
$1/|IL|I$
$X_L$
$Y_L$
$Z_L$
CALCULATE L
CALCULATE $f_{cp}$
$K_1$
$K_2$
+
+
CALCULATE DIFFUSE 2
+
OUT-IMAGE SEQUENTIAL WRITE ITERATOR
IMAGE SEQUENTIAL READ ITERATOR
CALCULATE SPECULAR 2
OUT-IMAGE SEQUENTIAL READ ITERATOR
CALCULATE AMBIENT
2ND AND SUBSEQUENT LIGHTS

Fig. 109

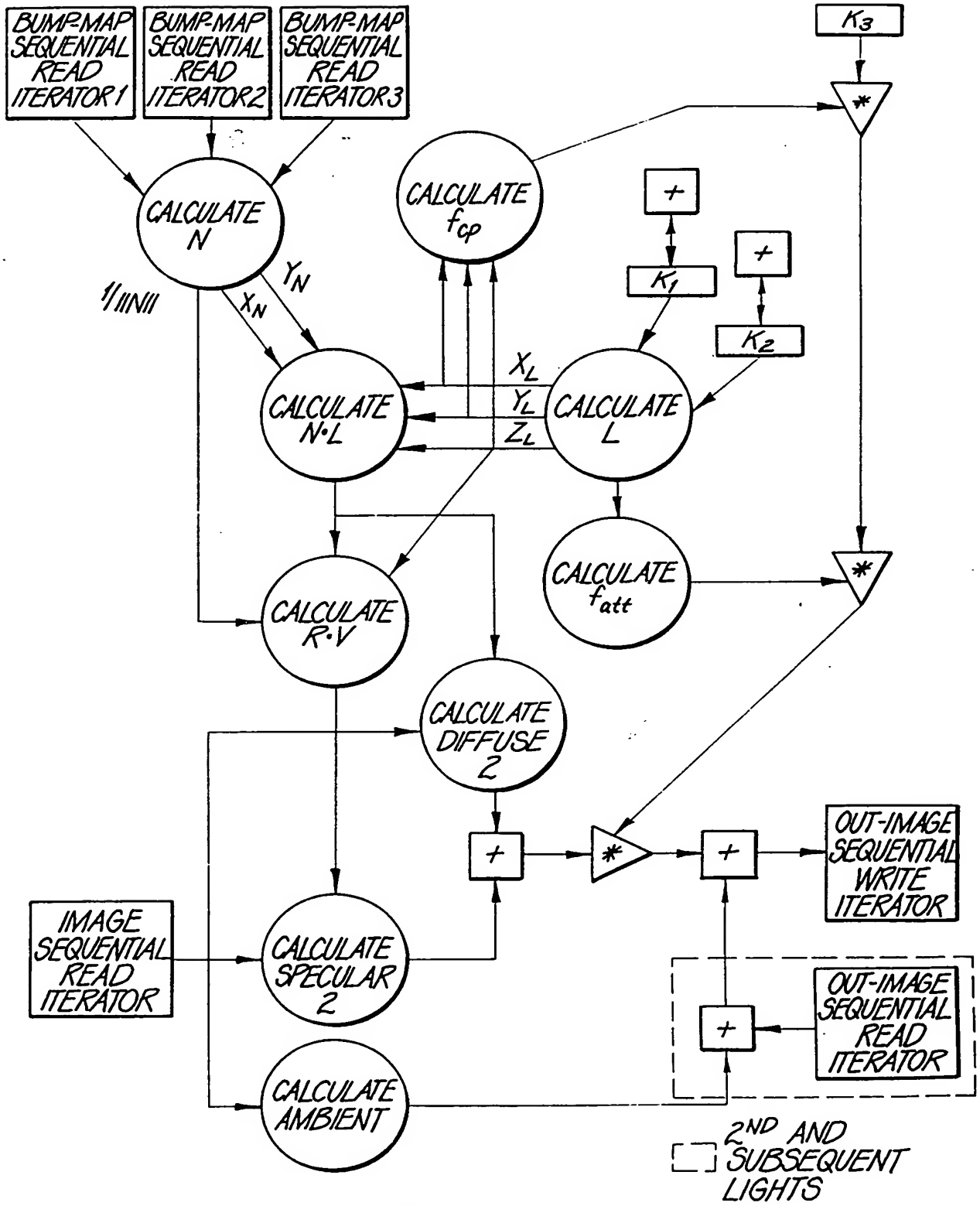

BUMP-MAP SEQUENTIAL READ ITERATOR 1
BUMP-MAP SEQUENTIAL READ ITERATOR 2
BUMP-MAP SEQUENTIAL READ ITERATOR 3
K3
CALCULATE N
CALCULATE fcp
K1
K2
YN
XN
XL
YL
ZL
CALCULATE N·L
CALCULATE L
CALCULATE fatt
CALCULATE R·V
CALCULATE DIFFUSE 2
OUT-IMAGE SEQUENTIAL WRITE ITERATOR
IMAGE SEQUENTIAL READ ITERATOR
CALCULATE SPECULAR 2
OUT-IMAGE SEQUENTIAL READ ITERATOR
CALCULATE AMBIENT
2ND AND SUBSEQUENT LIGHTS

Fig. 110

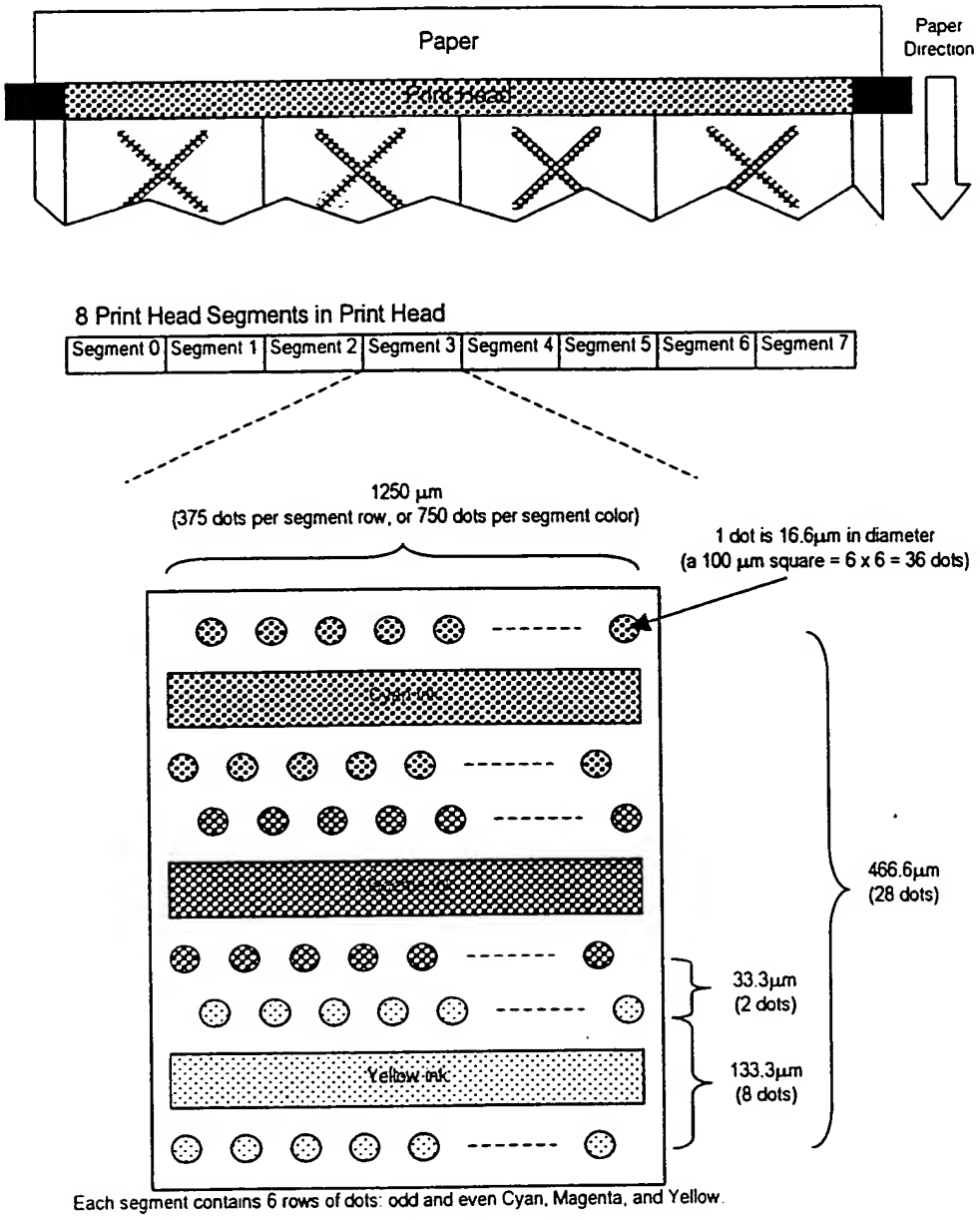

Paper
Paper Direction
Print Head
8 Print Head Segments in Print Head
Segment 0
Segment 1
Segment 2
Segment 3
Segment 4
Segment 5
Segment 6
Segment 7
1250 µm
(375 dots per segment row, or 750 dots per segment color)
1 dot is 16.6µm in diameter
(a 100 µm square = 6 x 6 = 36 dots)
Cyan ink
Magenta ink
Yellow ink
466.6µm
(28 dots)
33.3µm
(2 dots)
133.3µm
(8 dots)
Each segment contains 6 rows of dots: odd and even Cyan, Magenta, and Yellow.

Fig. 111

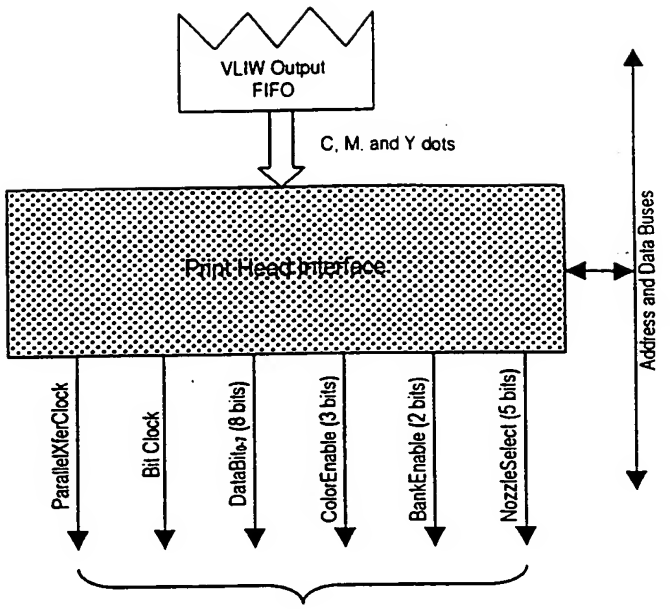



Fig. 112

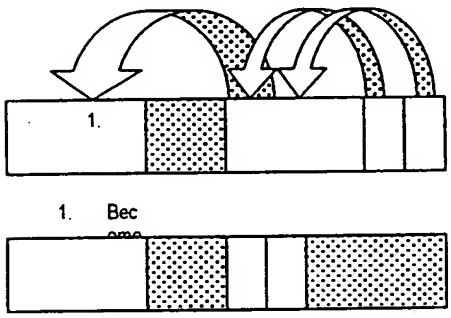



Fig. 113

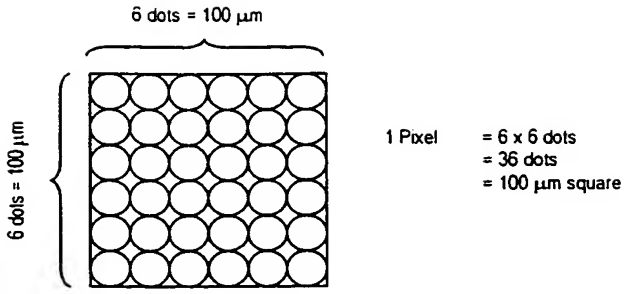



Fig. 114

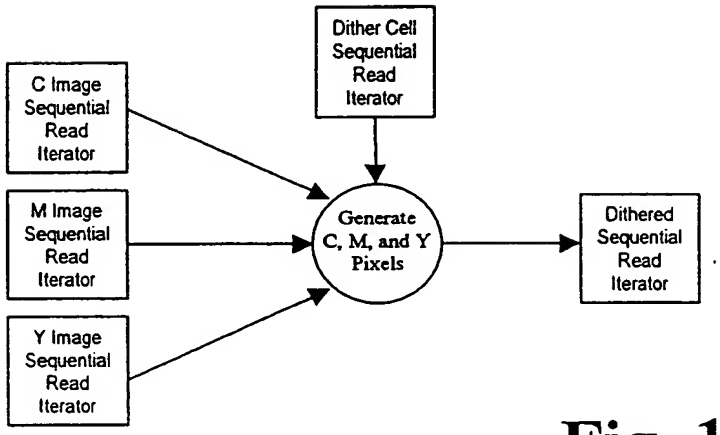



Fig. 115

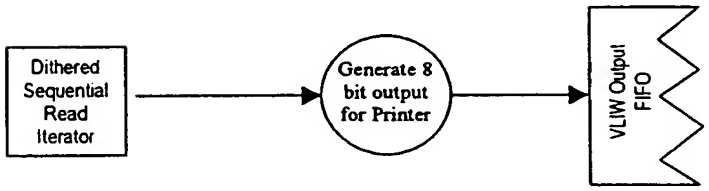



Fig. 116

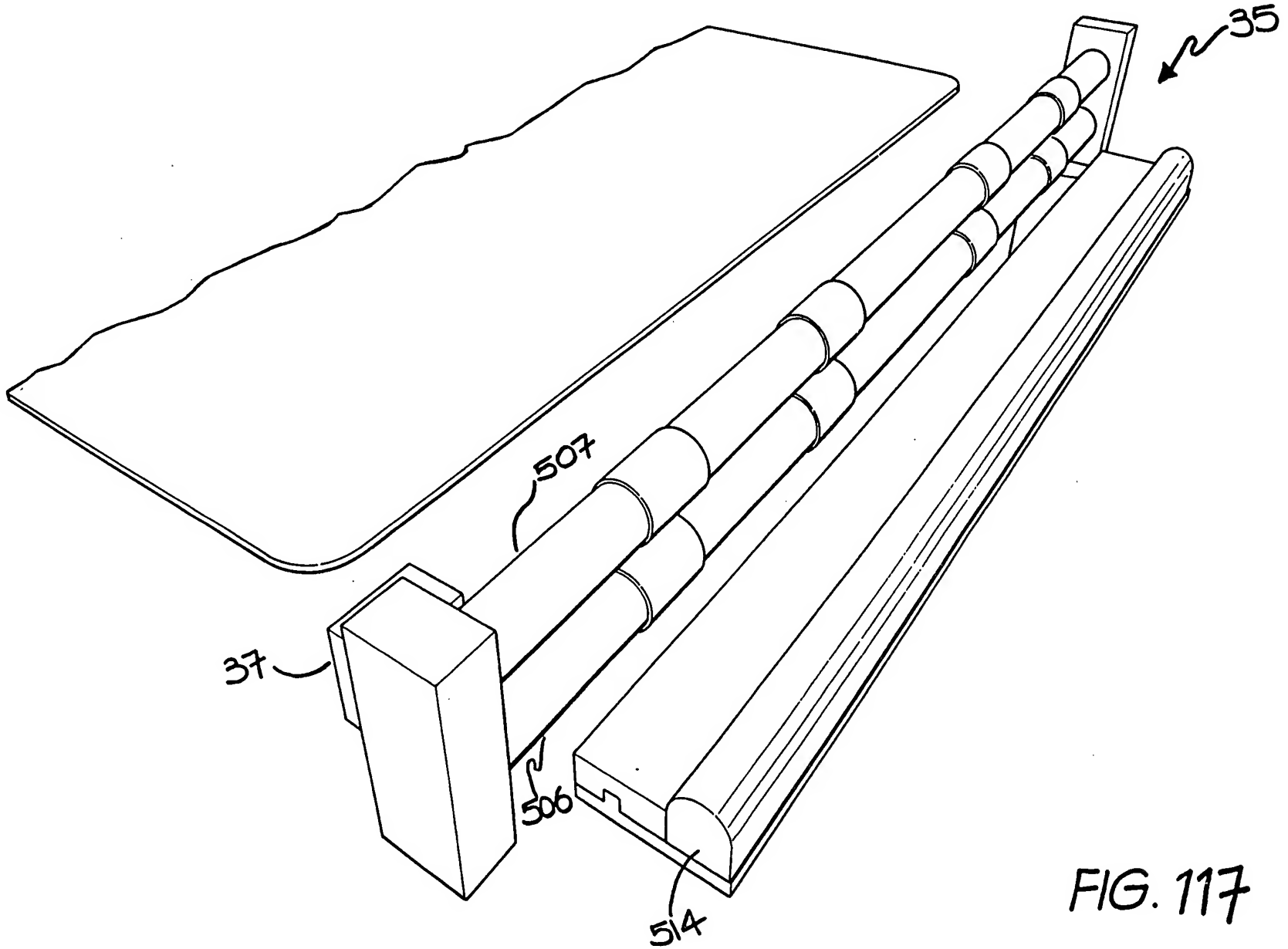



FIG. 117

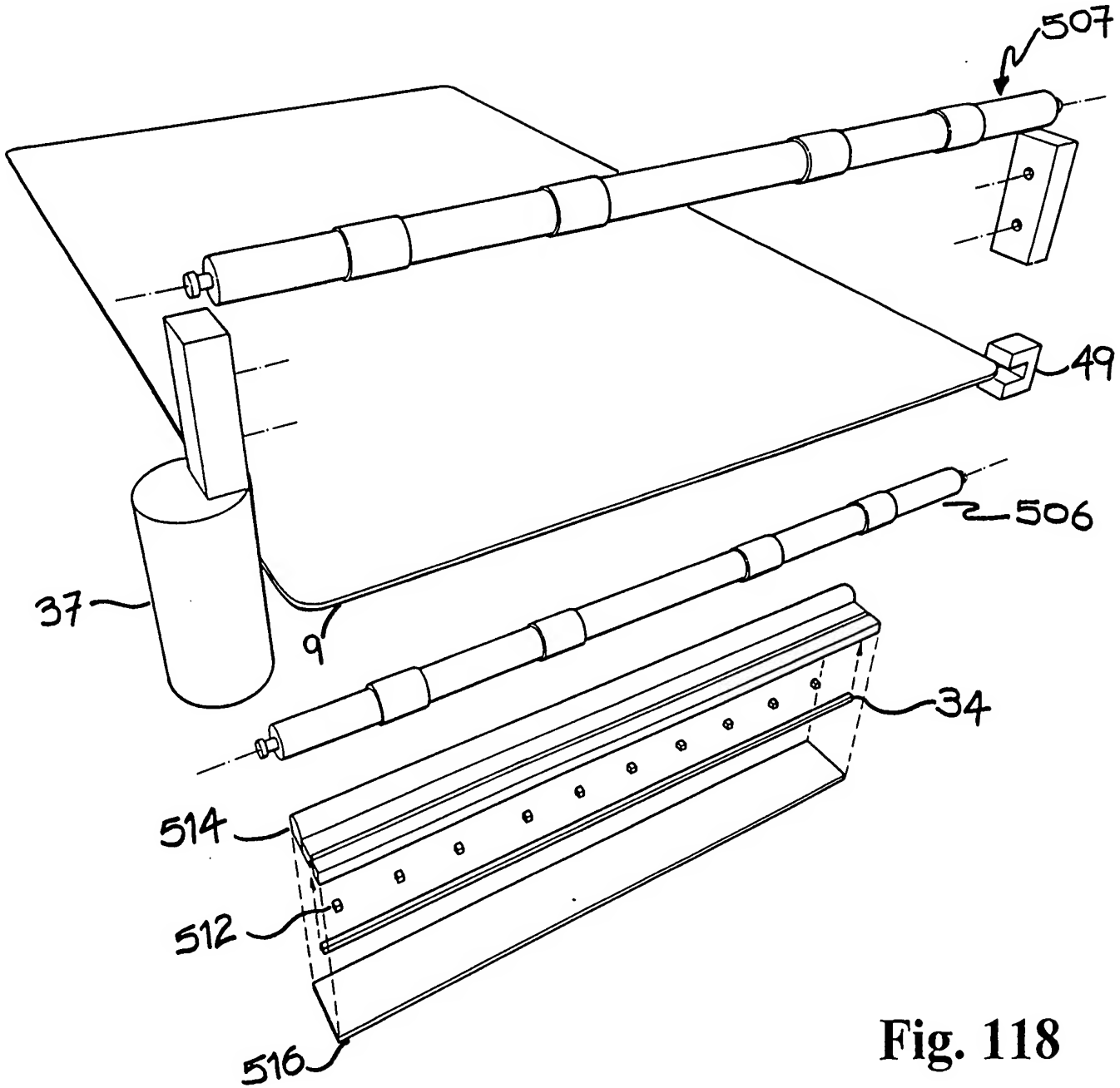



Fig. 118

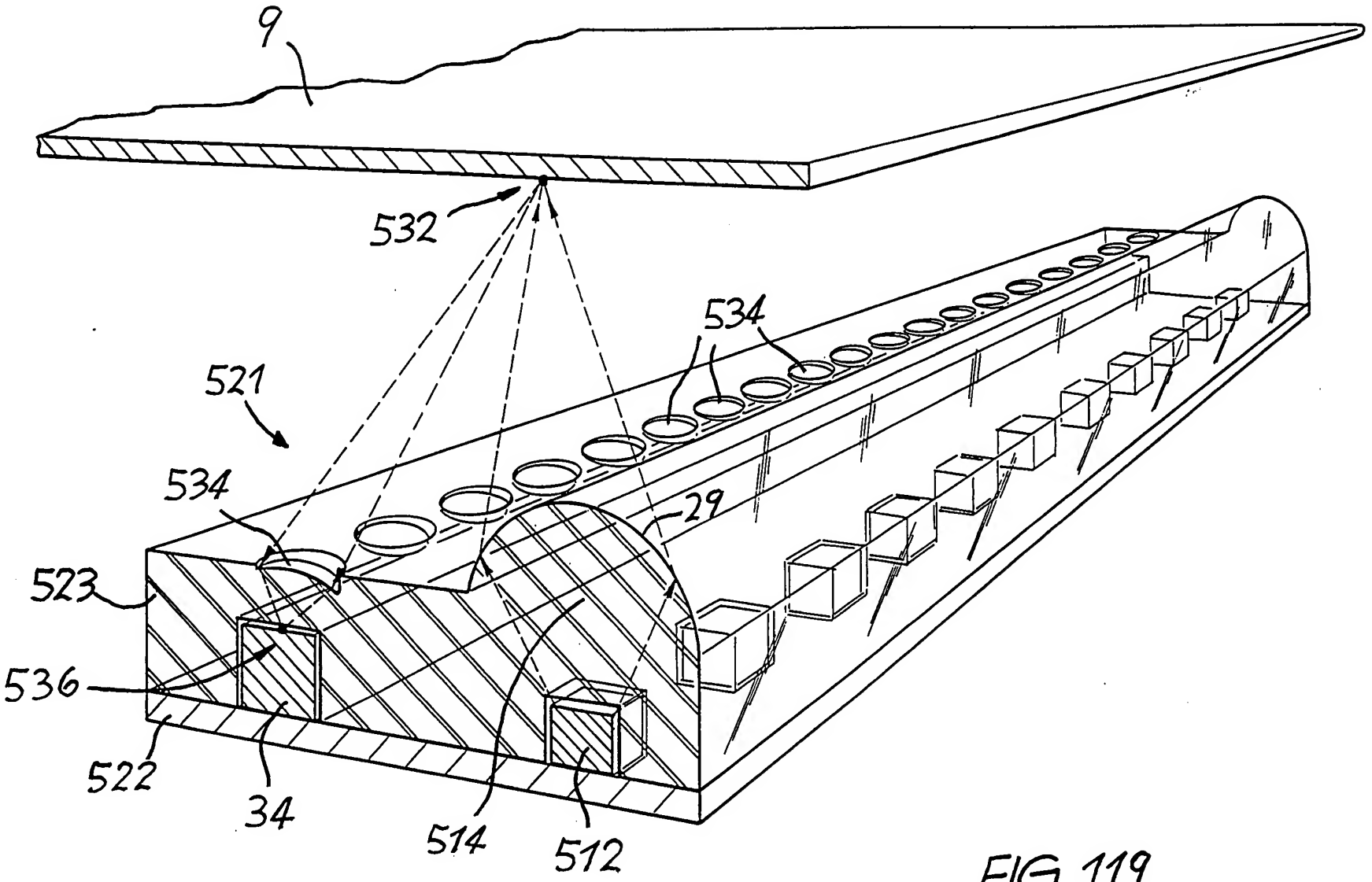



FIG. 119

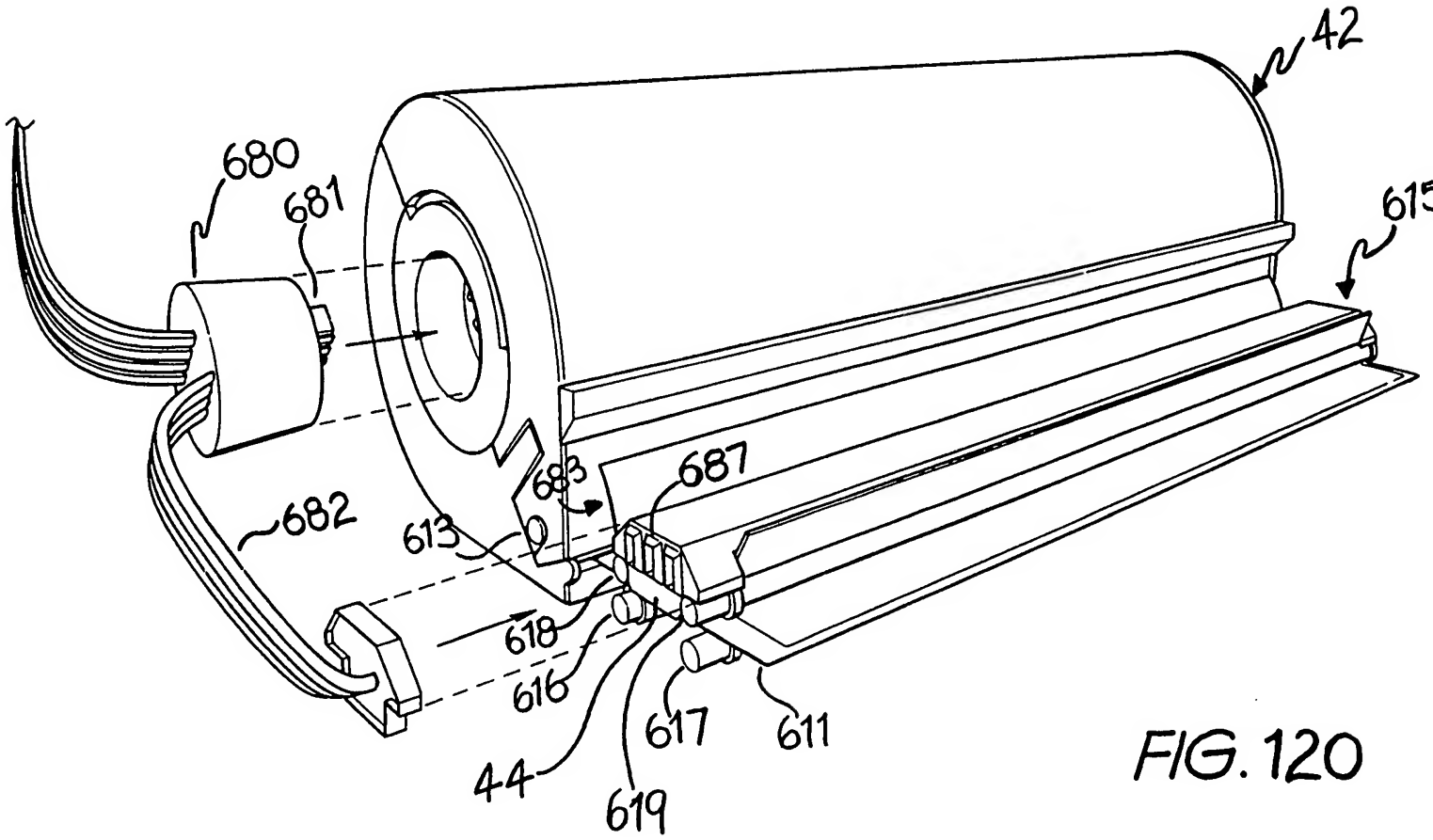



FIG. 120

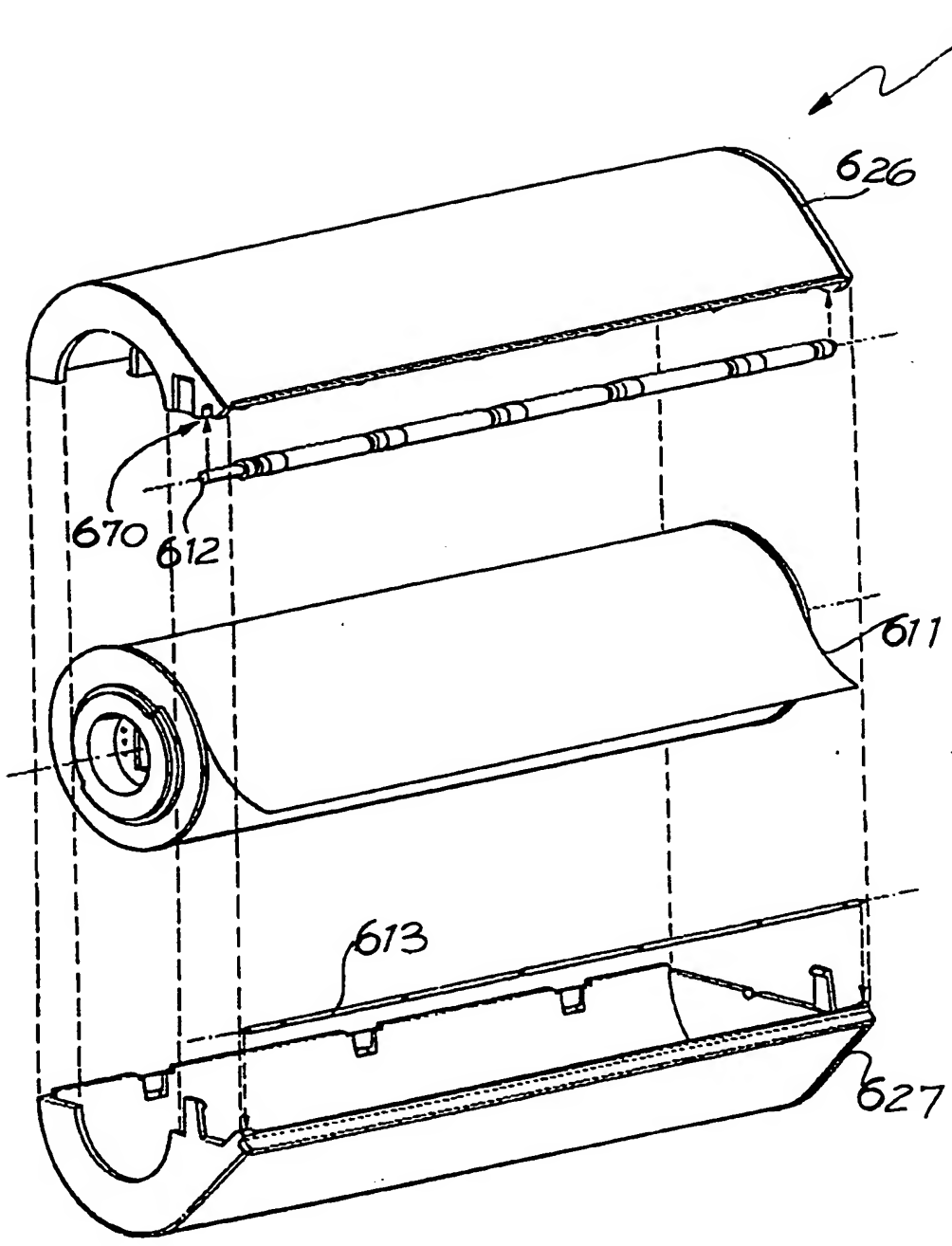



FIG. 121

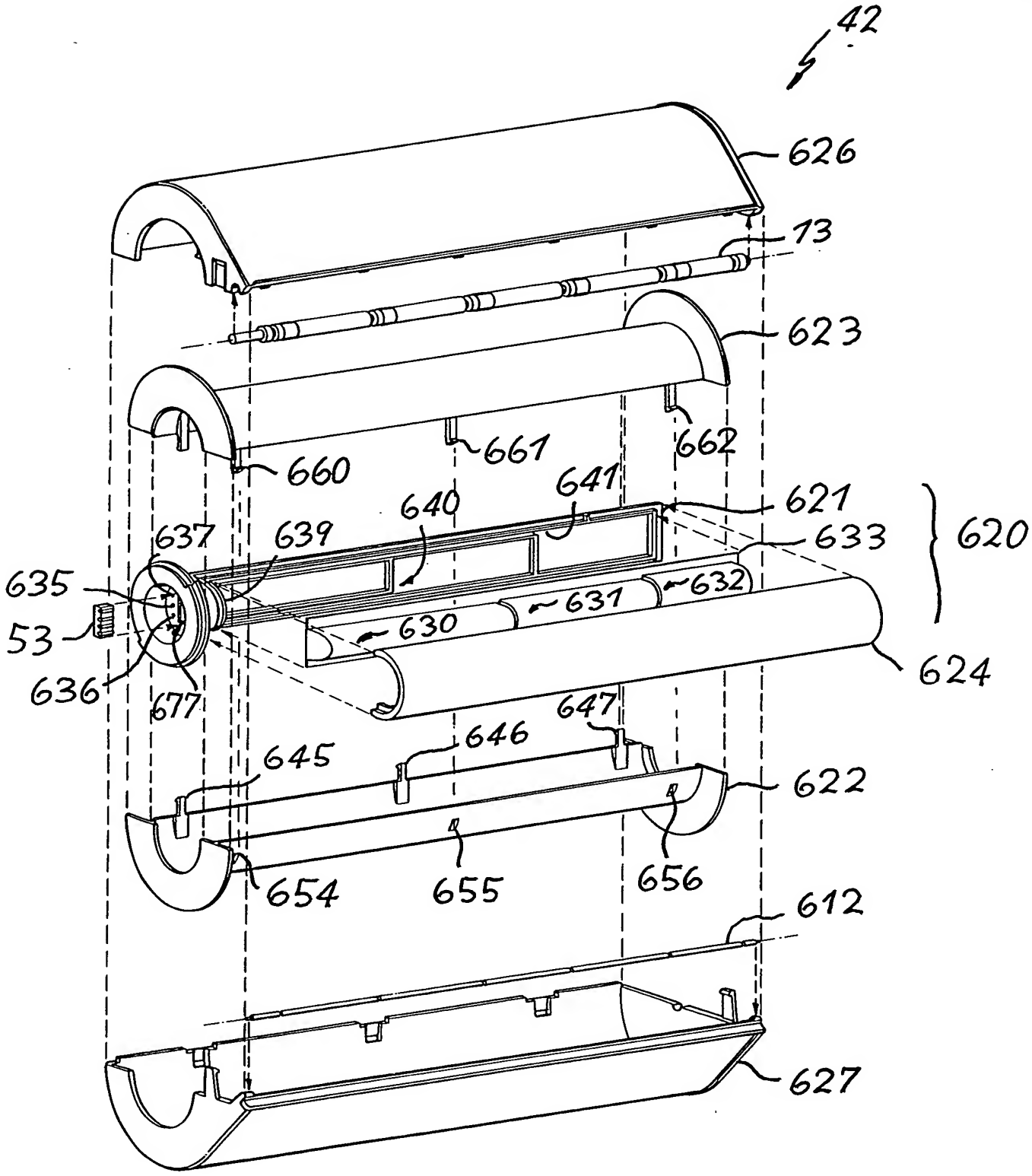

42
626
73
623
660
661
662
641
640
621
620
637
639
633
635
631
632
53
630
624
636
677
647
646
645
622
654
655
656
612
627

FIG. 122

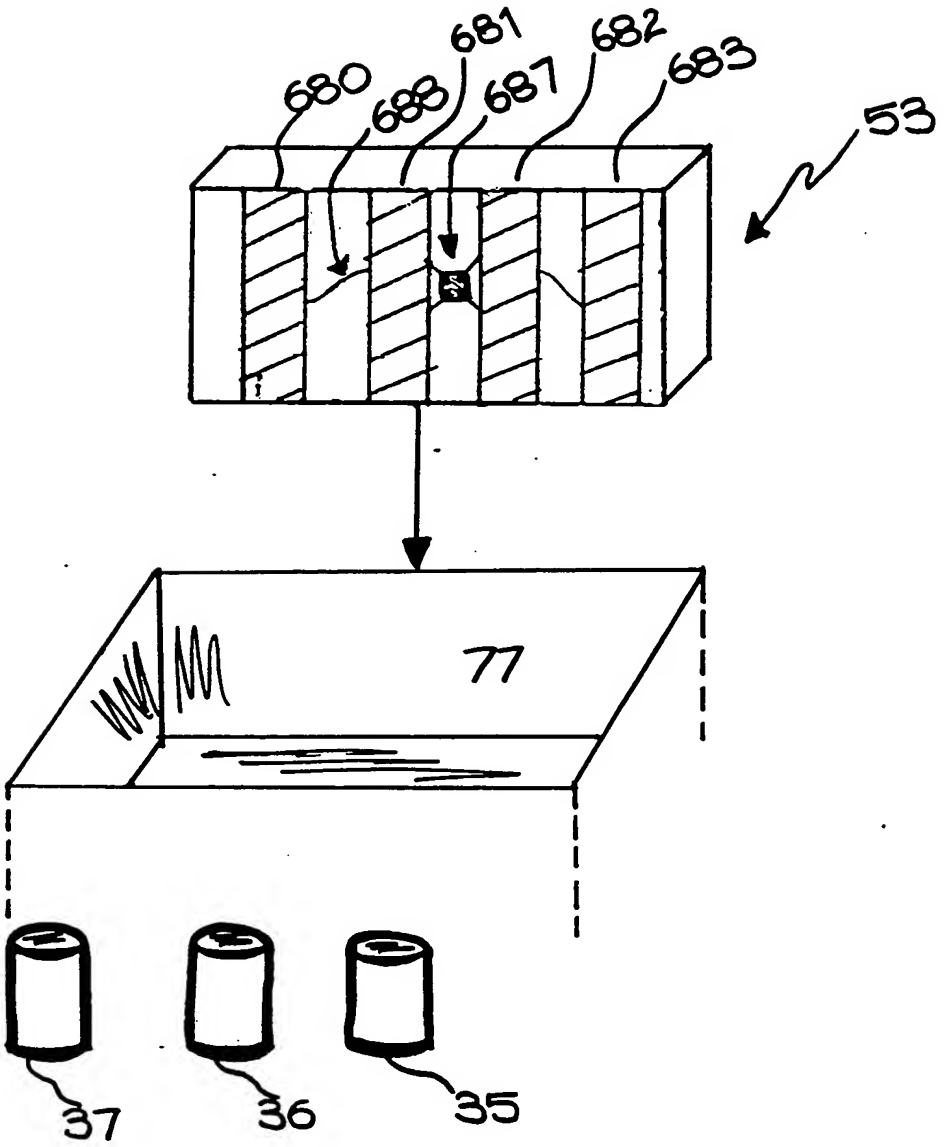



FIG. 123

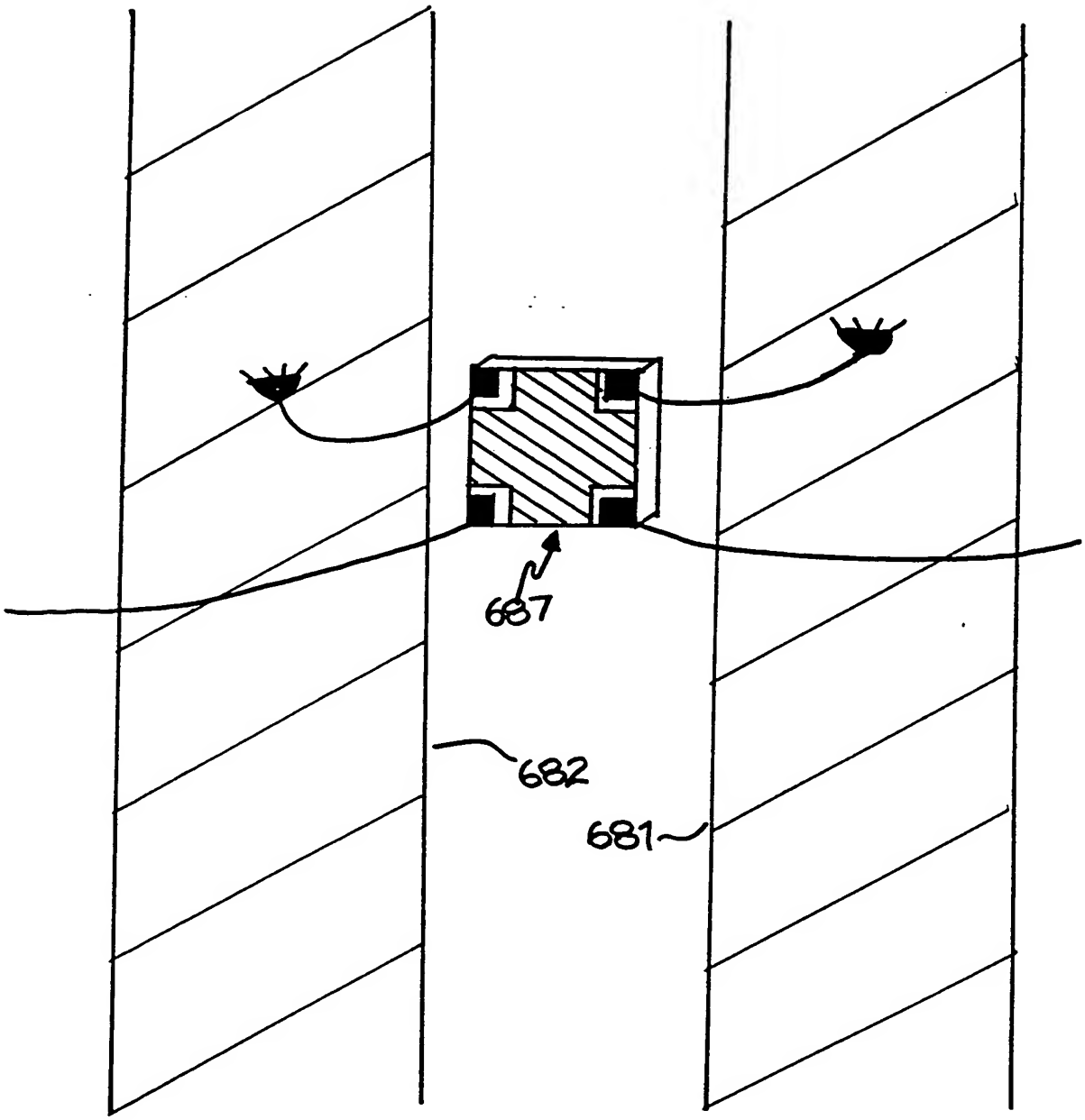



FIG. 124

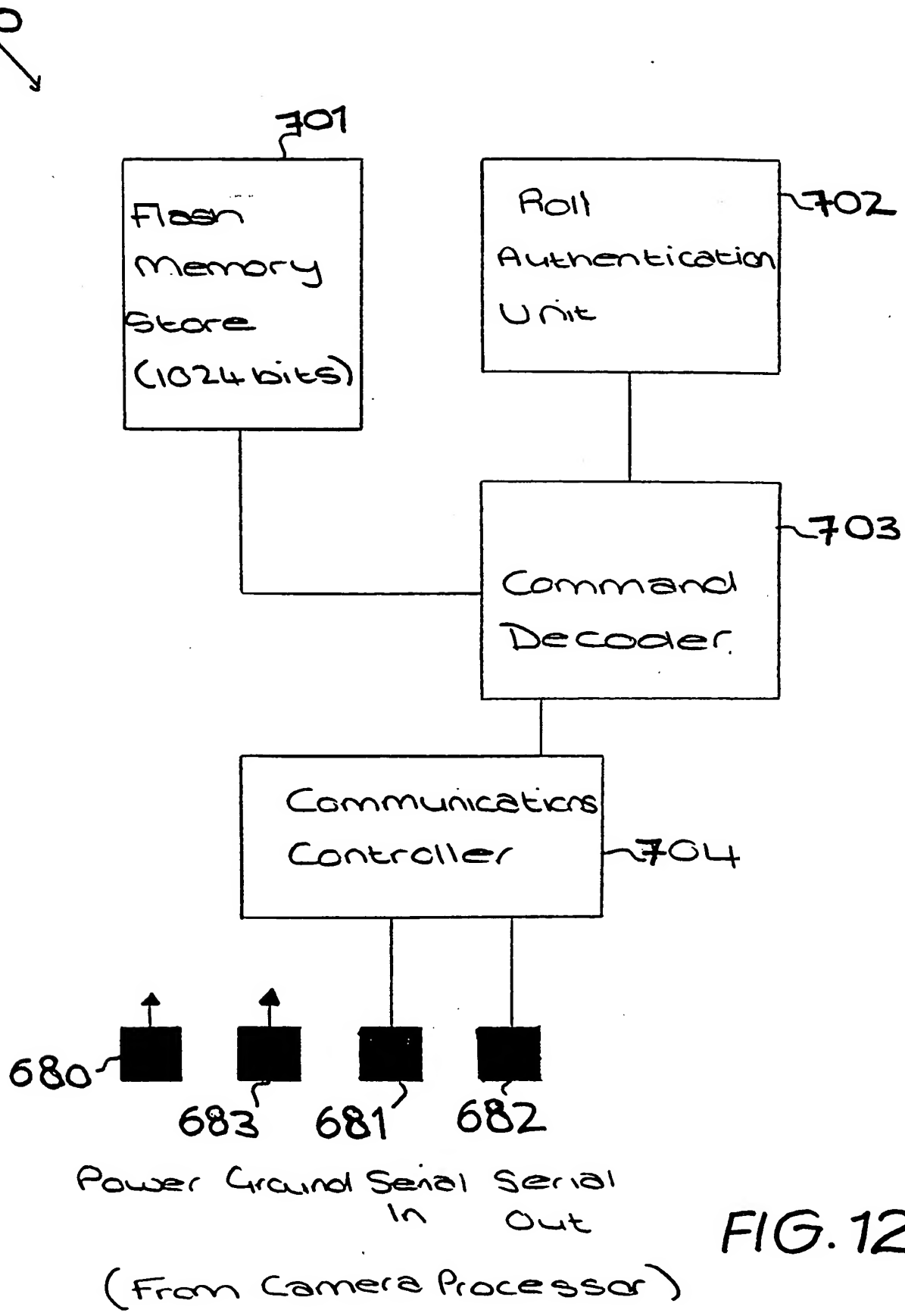

700
701
Flash Memory Store (1024 bits)
Roll Authentication Unit
702
703
Command Decoder
Communications Controller
704
680
683
681
682
Power
Ground
Serial In
Serial Out
(From Camera Processor)

FIG. 125

705

| Data Type | Bits |
|---|---|
| Factory code | 16 |
| Batch number | 32 |
| Serial number | 48 |
| Manufacturing date | 16 |
| Media length | 24 |
| Media type | 8 |
| Preprinted media length | 16 |
| Cyan ink viscosity | 8 |
| Magenta ink viscosity | 8 |
| Yellow ink viscosity | 8 |
| Cyan drop volume | 8 |
| Magenta drop volume | 8 |
| Yellow drop volume | 8 |
| Cyan ink color | 24 |
| Magenta ink color | 24 |
| Yellow ink color | 24 |
| Remaining-media length indicator | 16 |
| Authentication key | 128 |
| Copyrightable bit pattern | 512 |
| Reserved for camera use | 88 |
| **Total** | **1024** |

728

FIG. 126

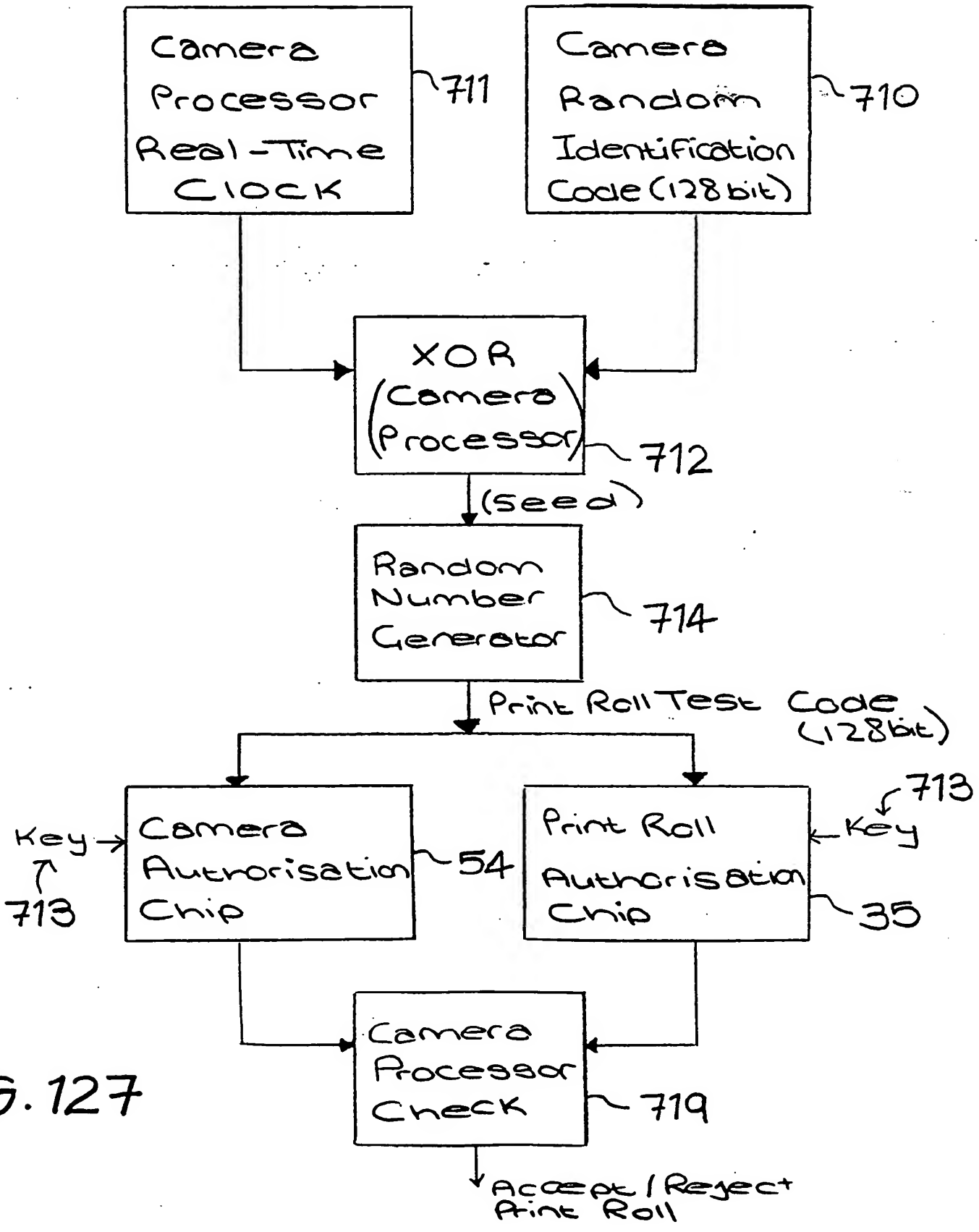

Camera Processor Real-Time Clock
711
Camera Random Identification Code (128 bit)
710
XOR (Camera Processor)
712
(Seed)
Random Number Generator
714
Print Roll Test Code (128 bit)
713
Key
Key
713
Camera Authorisation Chip
54
Print Roll Authorisation Chip
35
Camera Processor Check
719
Accept / Reject Print Roll

FIG. 127

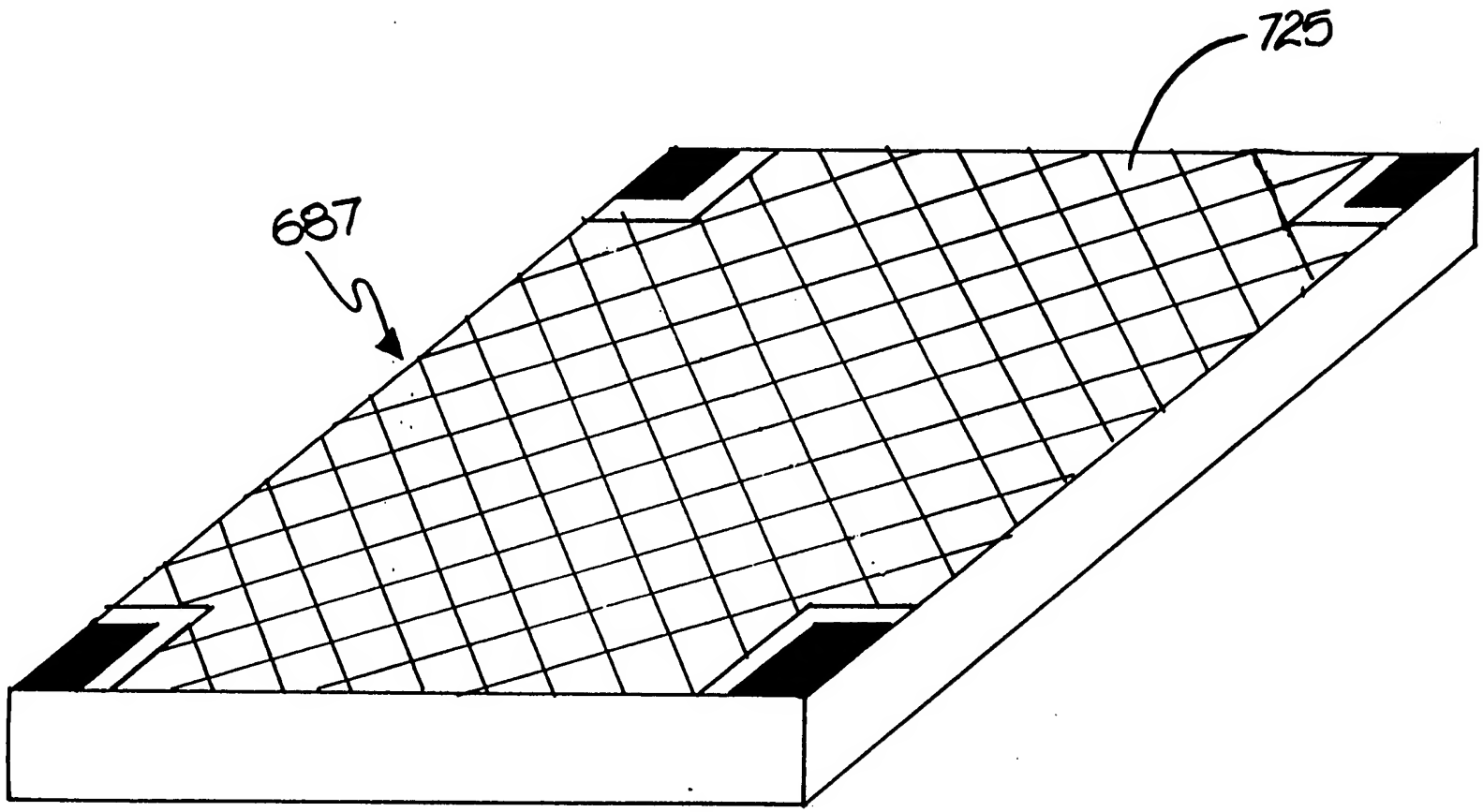



FIG. 128

730

| Data Type | Bits |
|---|---|
| Manufacturer code | 16 |
| Batch number | 32 |
| Serial number | 32 |
| Manufacturing date | 16 |
| Print engine type | 8 |
| Print resolution | 16 |
| Print counter | 16 |
| Authentication test key (random) | 128 |
| Print roll Authentication key | 128 |
| Bit pattern | 512 |
| Spare for camera use | 120 |
| **Total** | **1024** |

713

**Fig. 129**

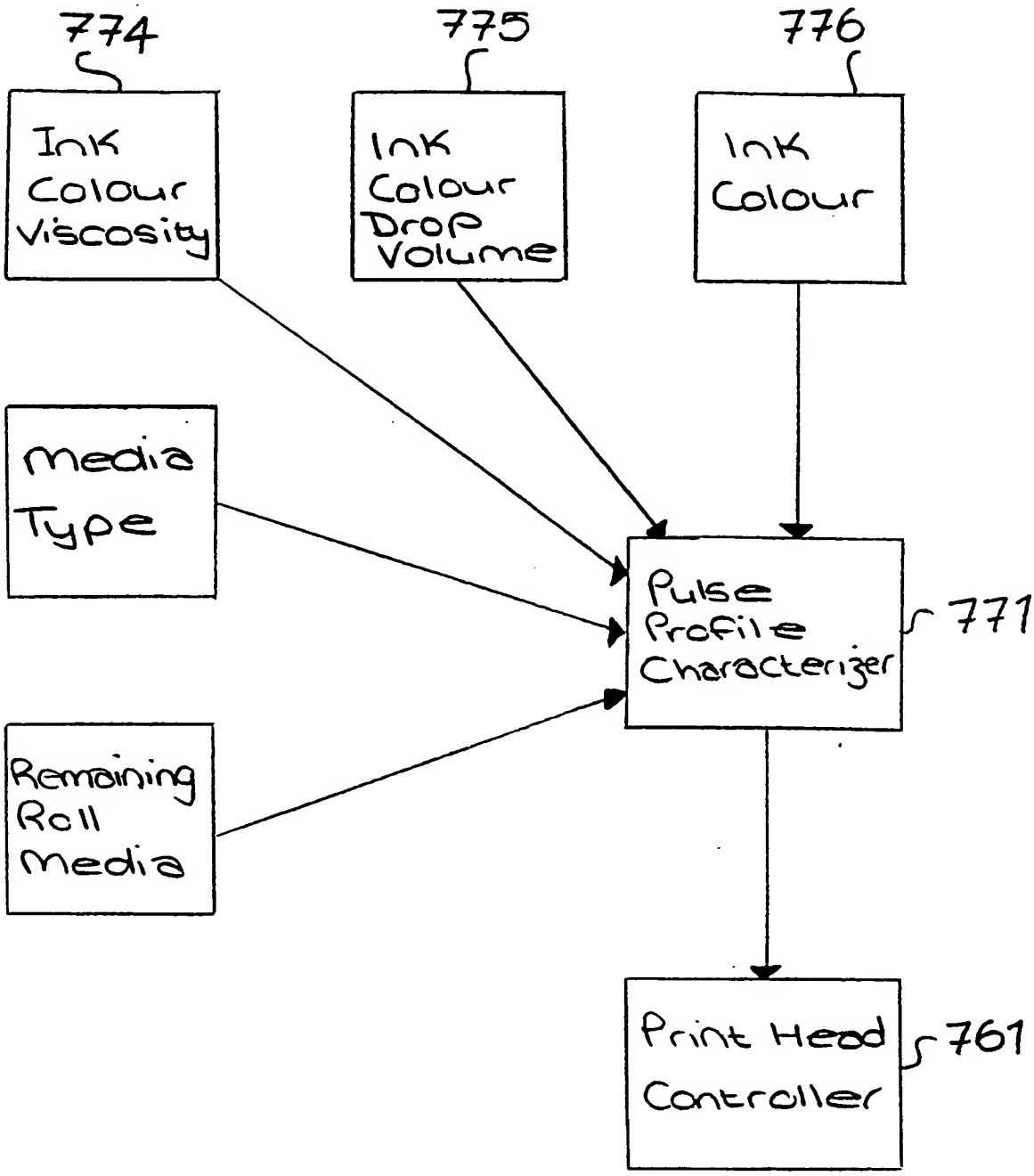

774
775
776
Ink Colour Viscosity
Ink Colour Drop Volume
Ink Colour
Media Type
Remaining Roll Media
Pulse Profile Characterizer
771
Print Head Controller
761

FIG.13O

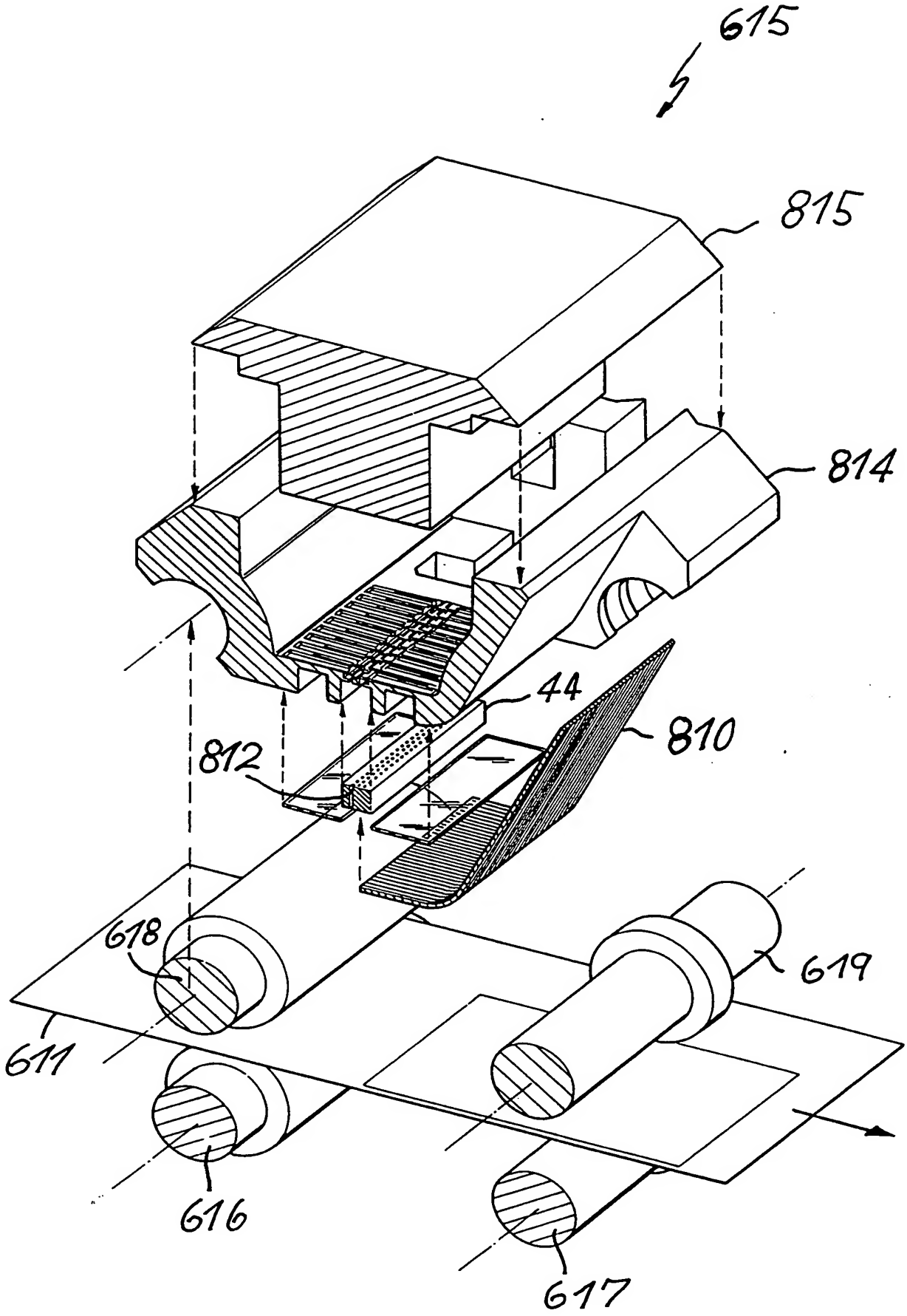



**Fig. 131**

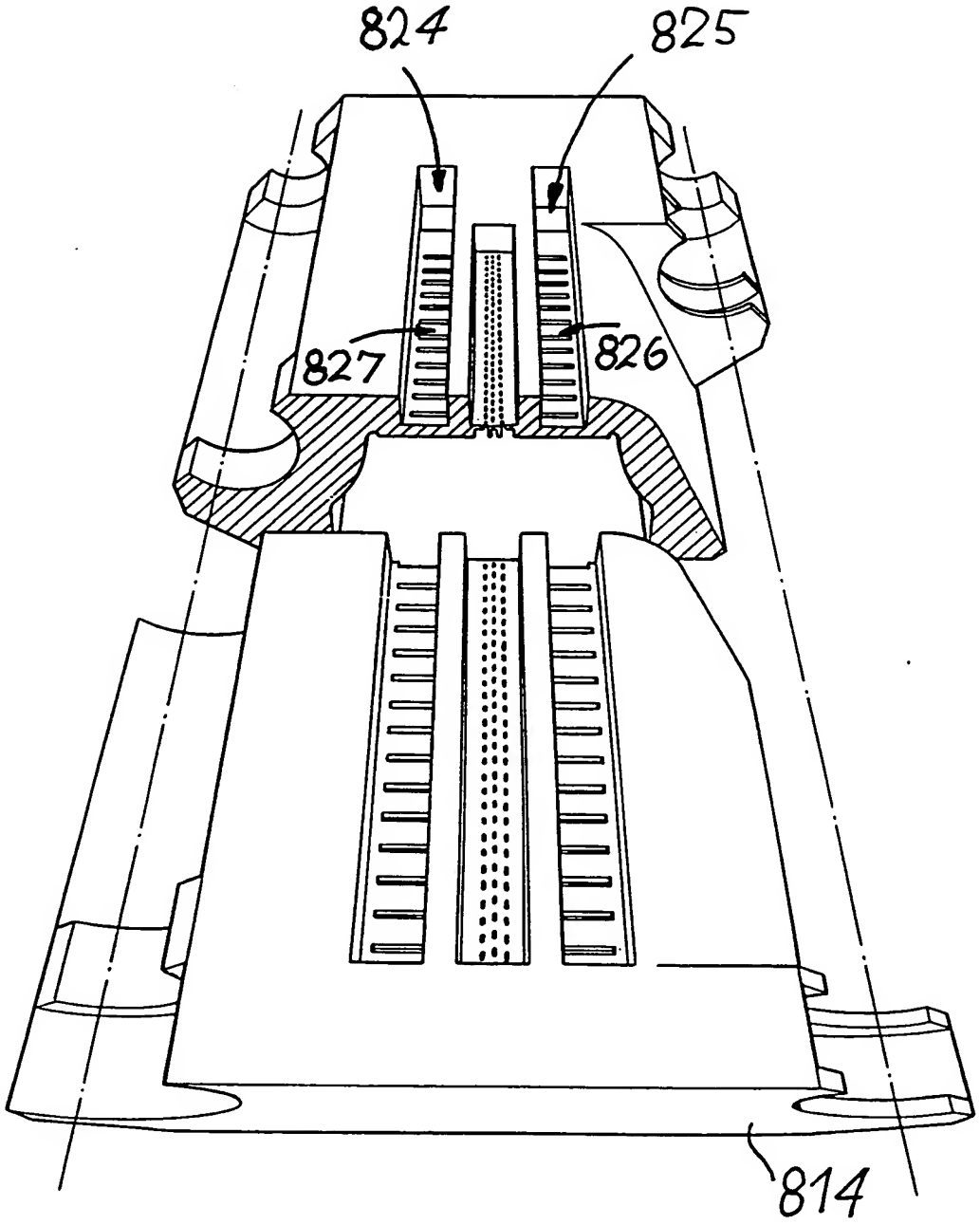



Fig. 132

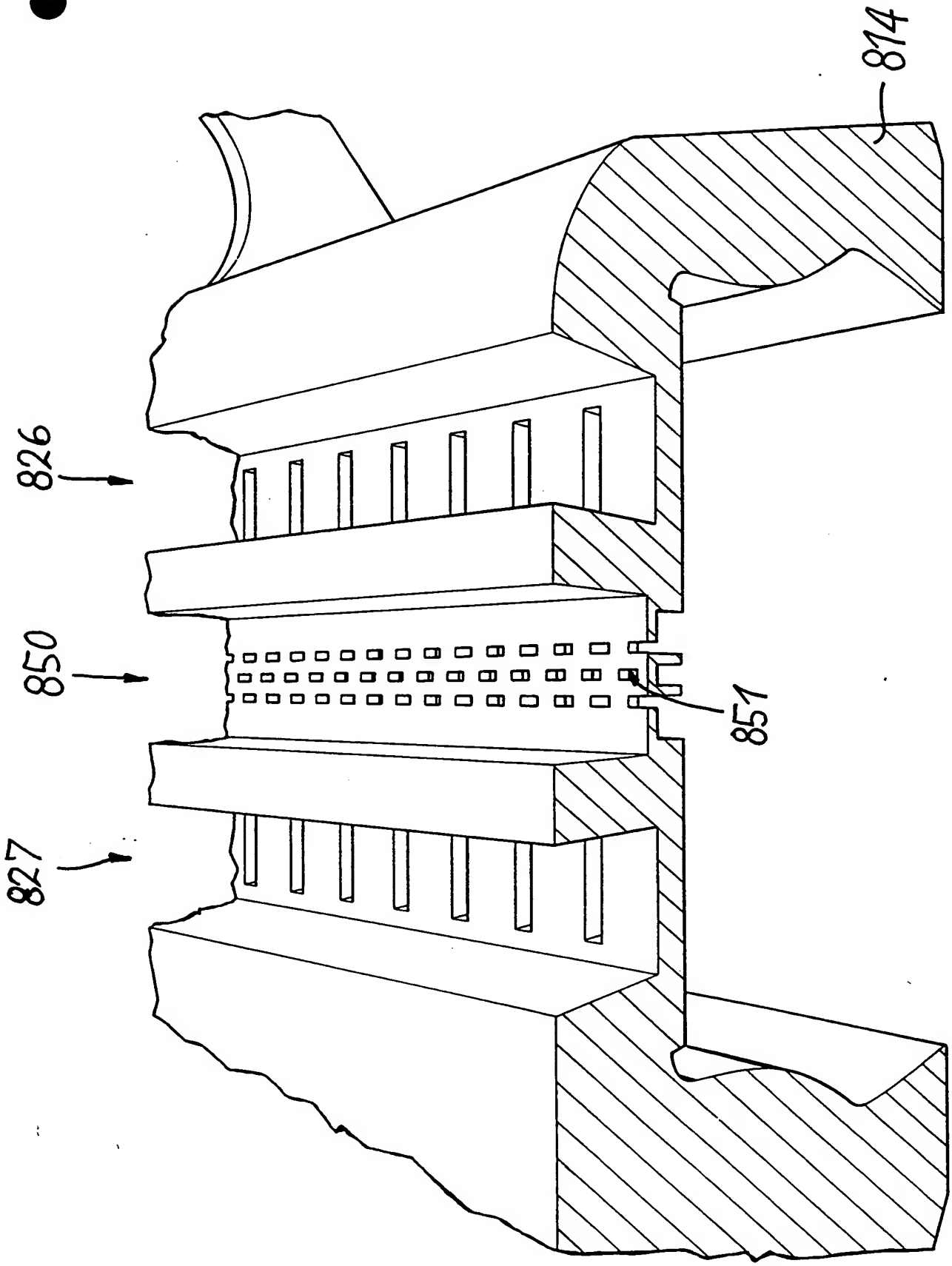



Fig. 133

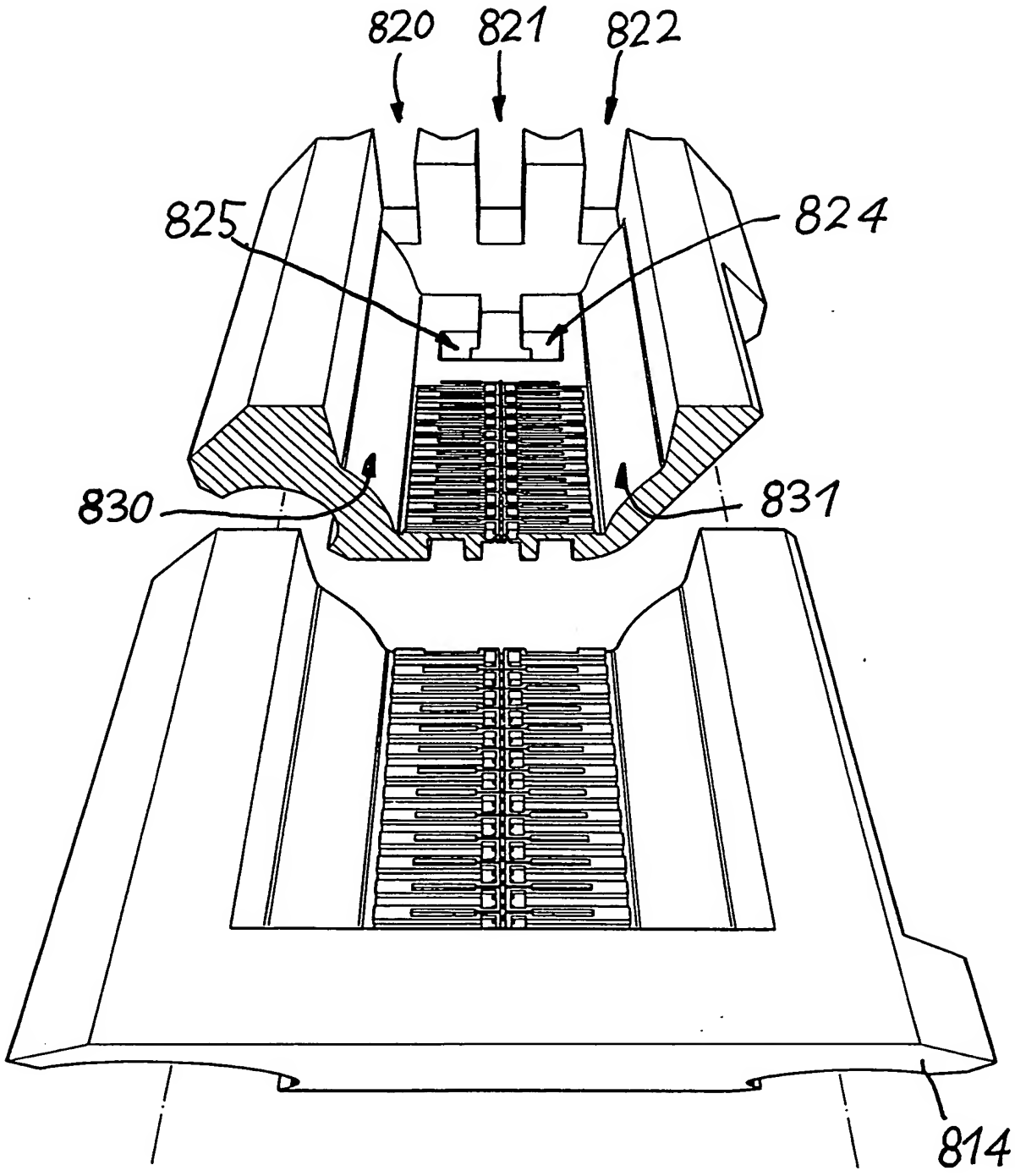



Fig. 134

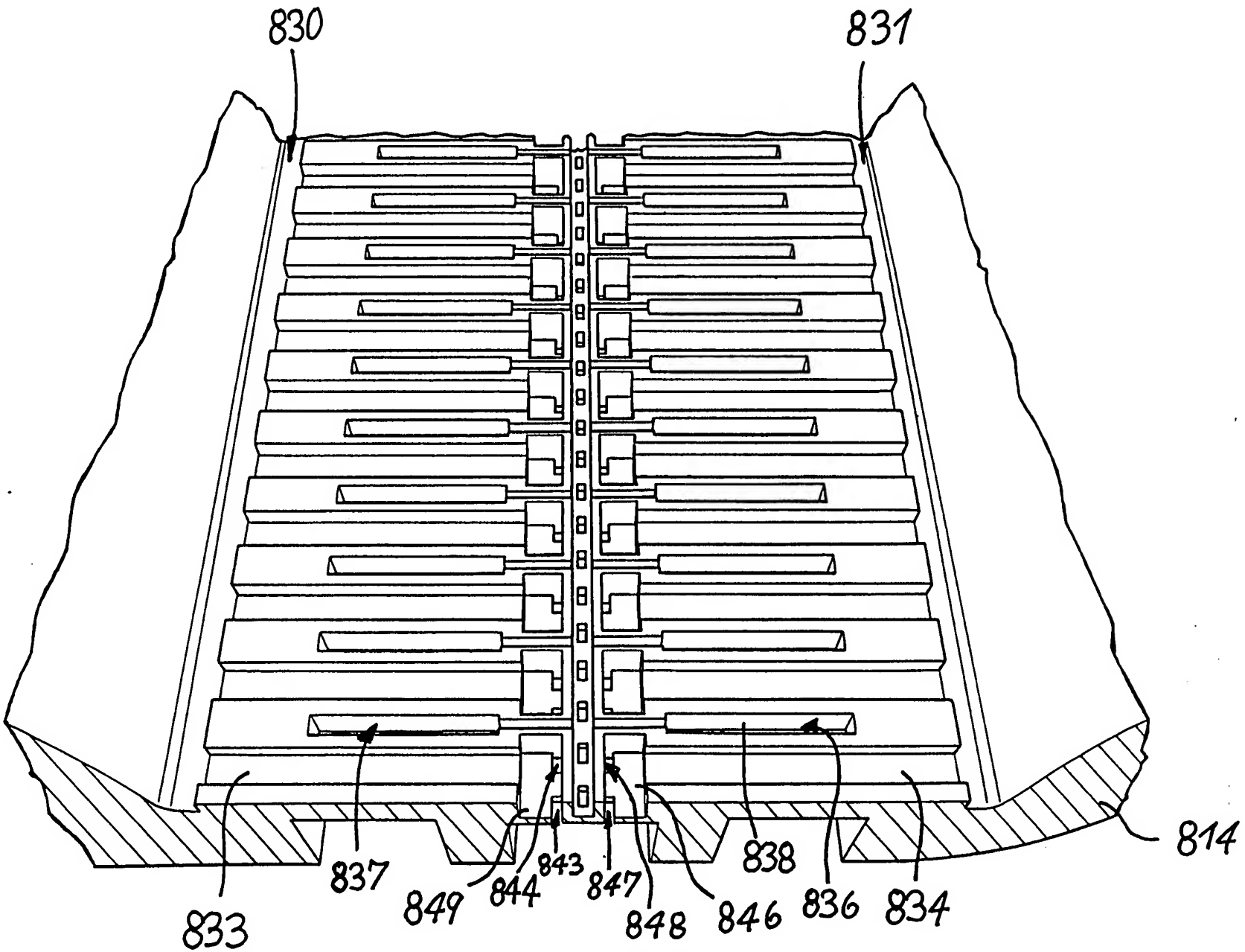



Fig. 135

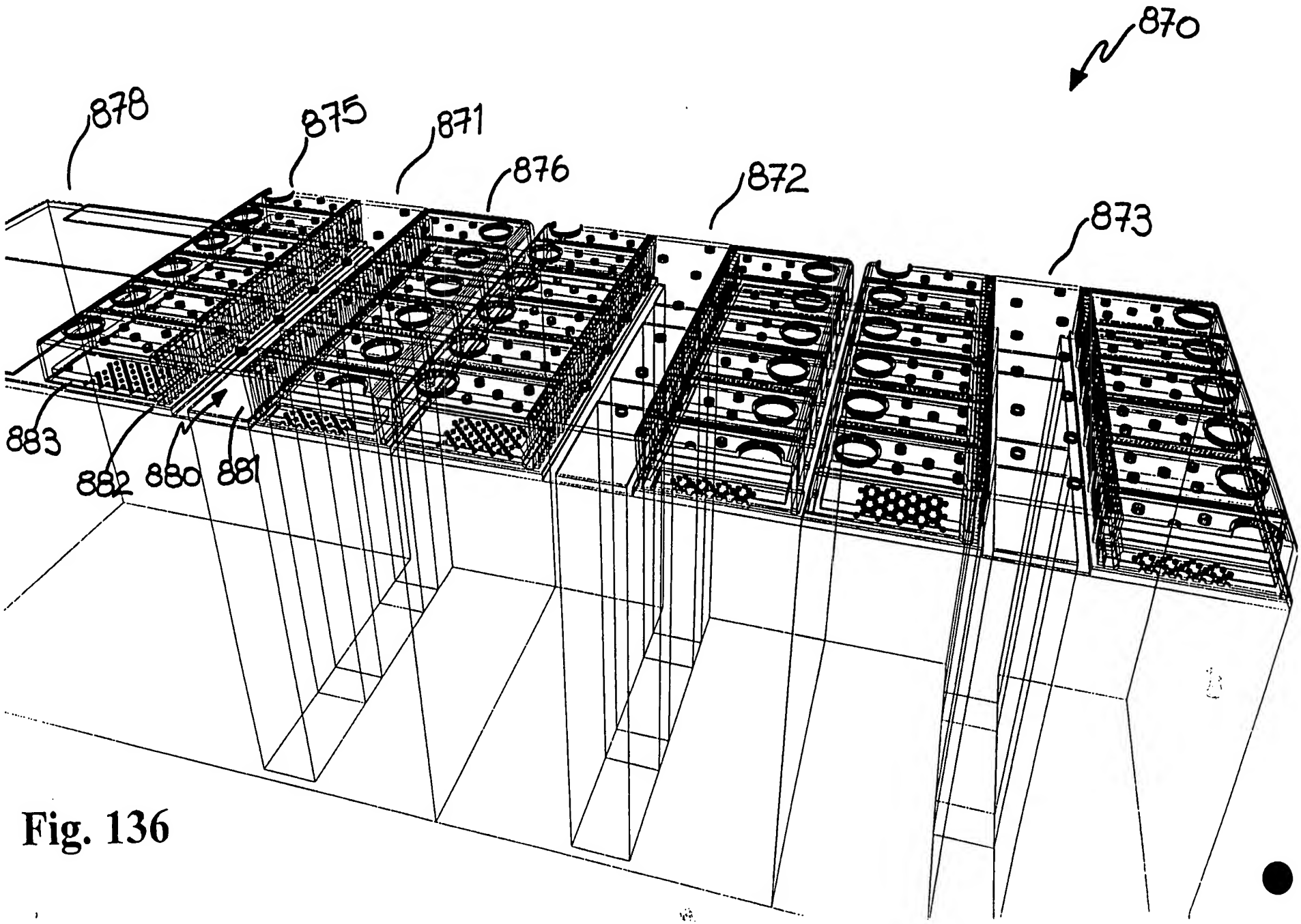



Fig. 136

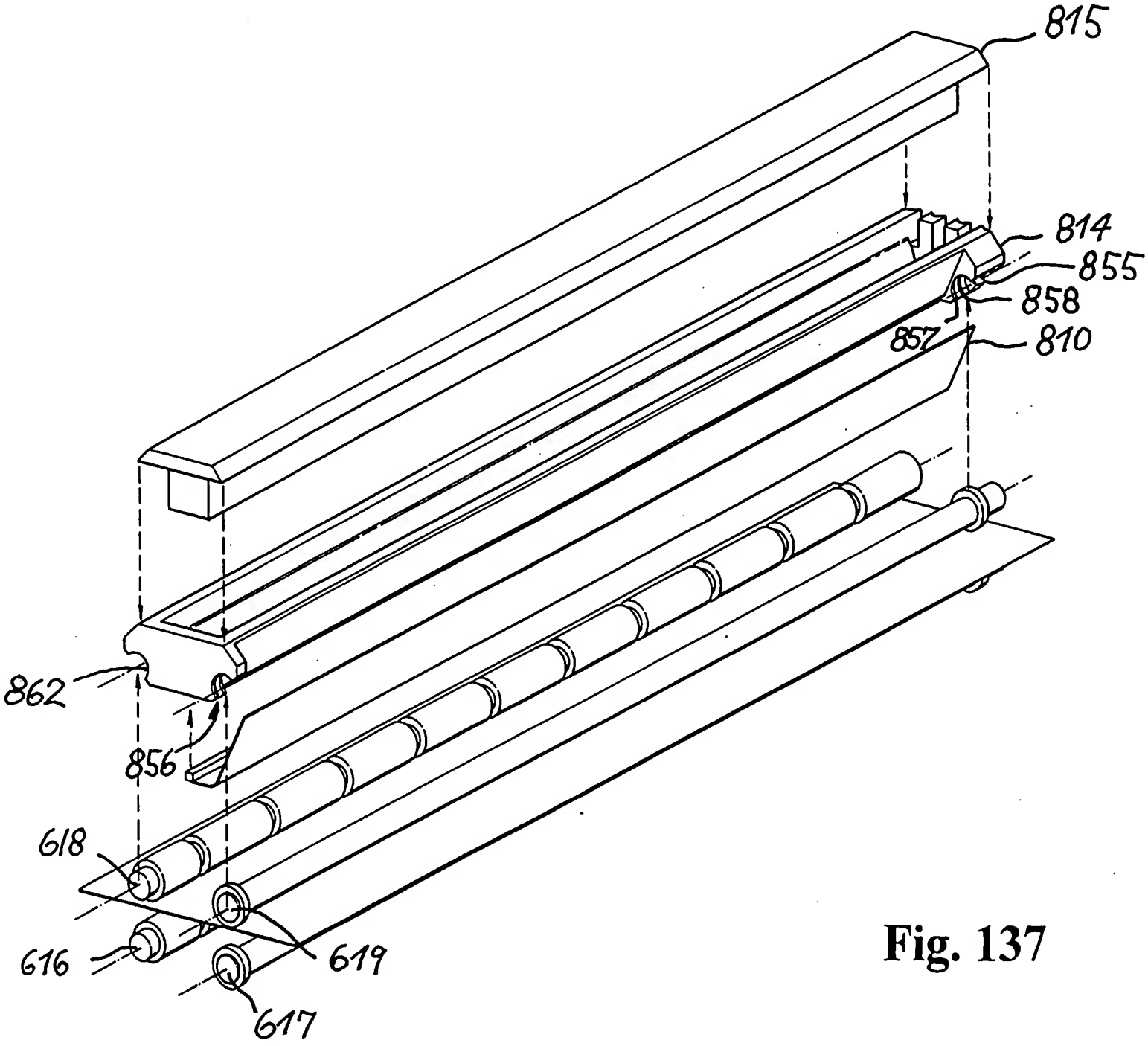



Fig. 137

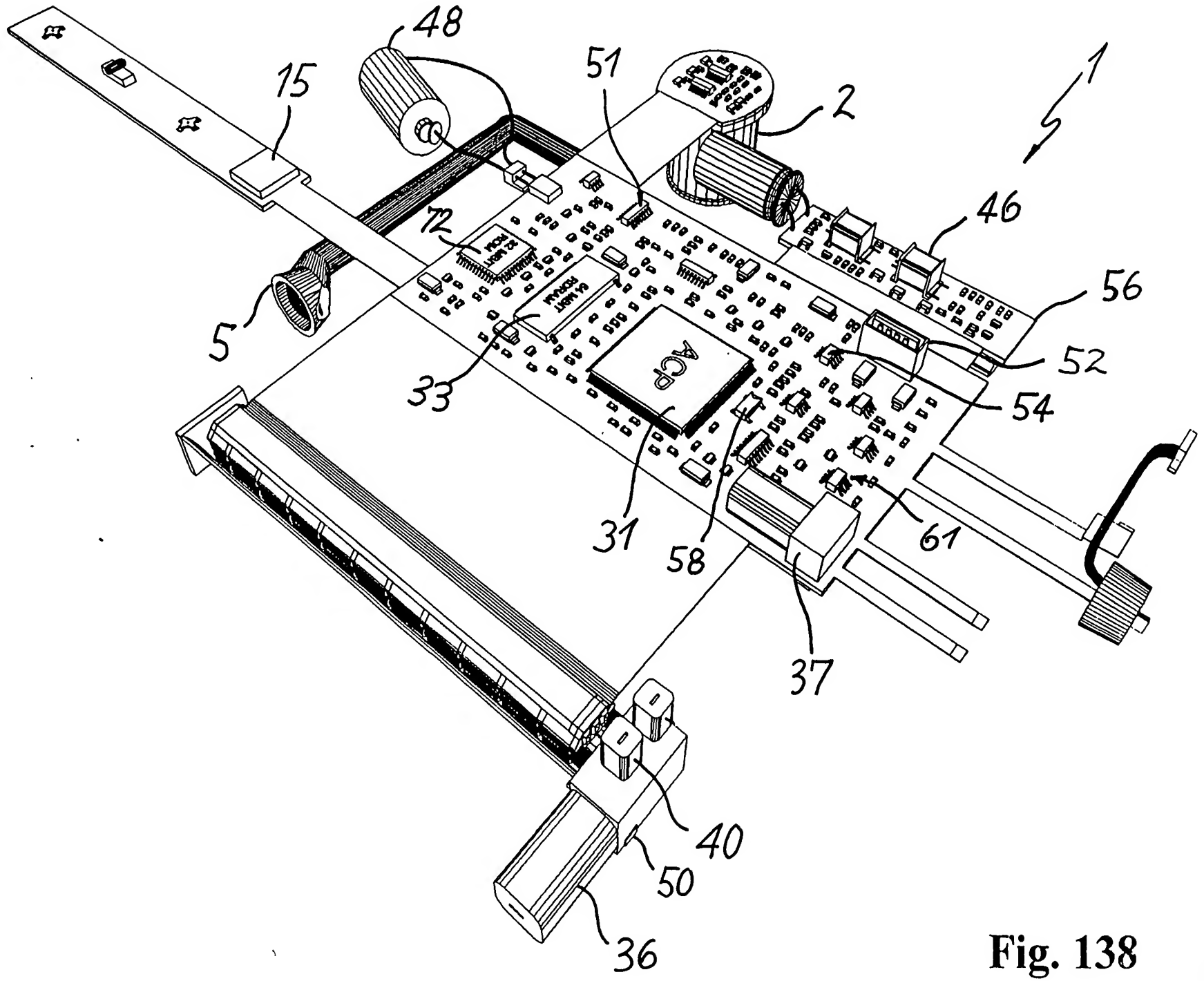



Fig. 138

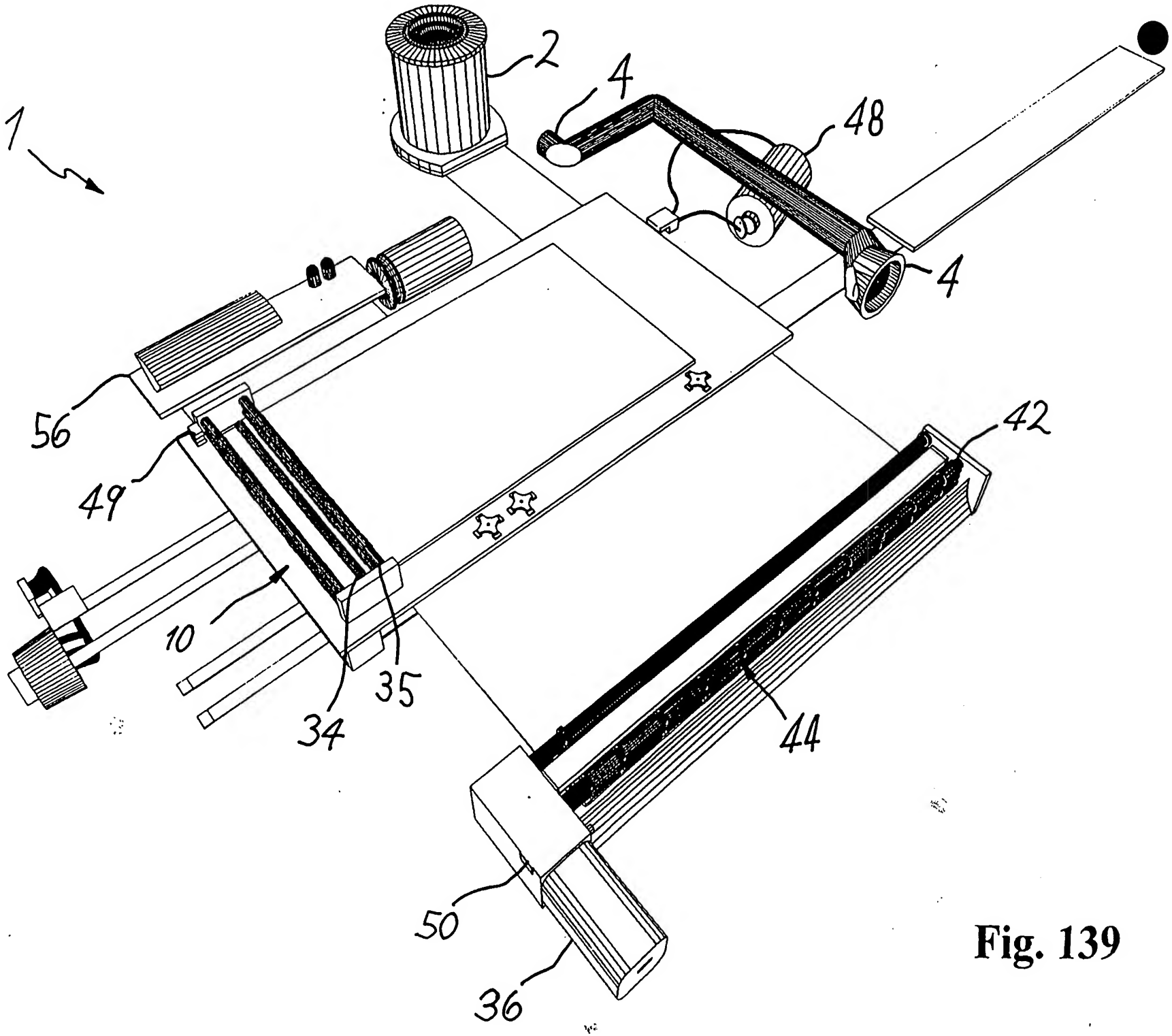



Fig. 139

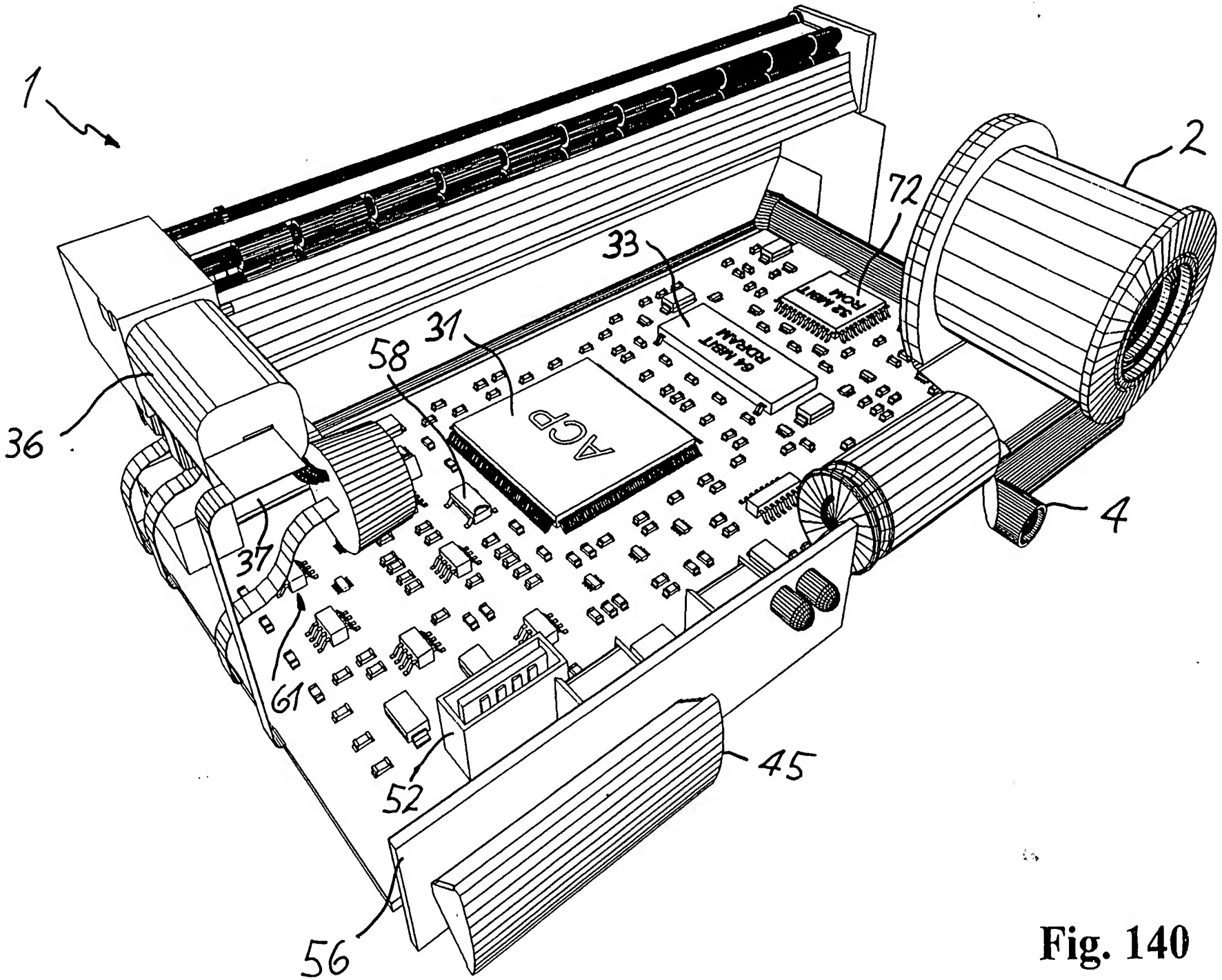



Fig. 140

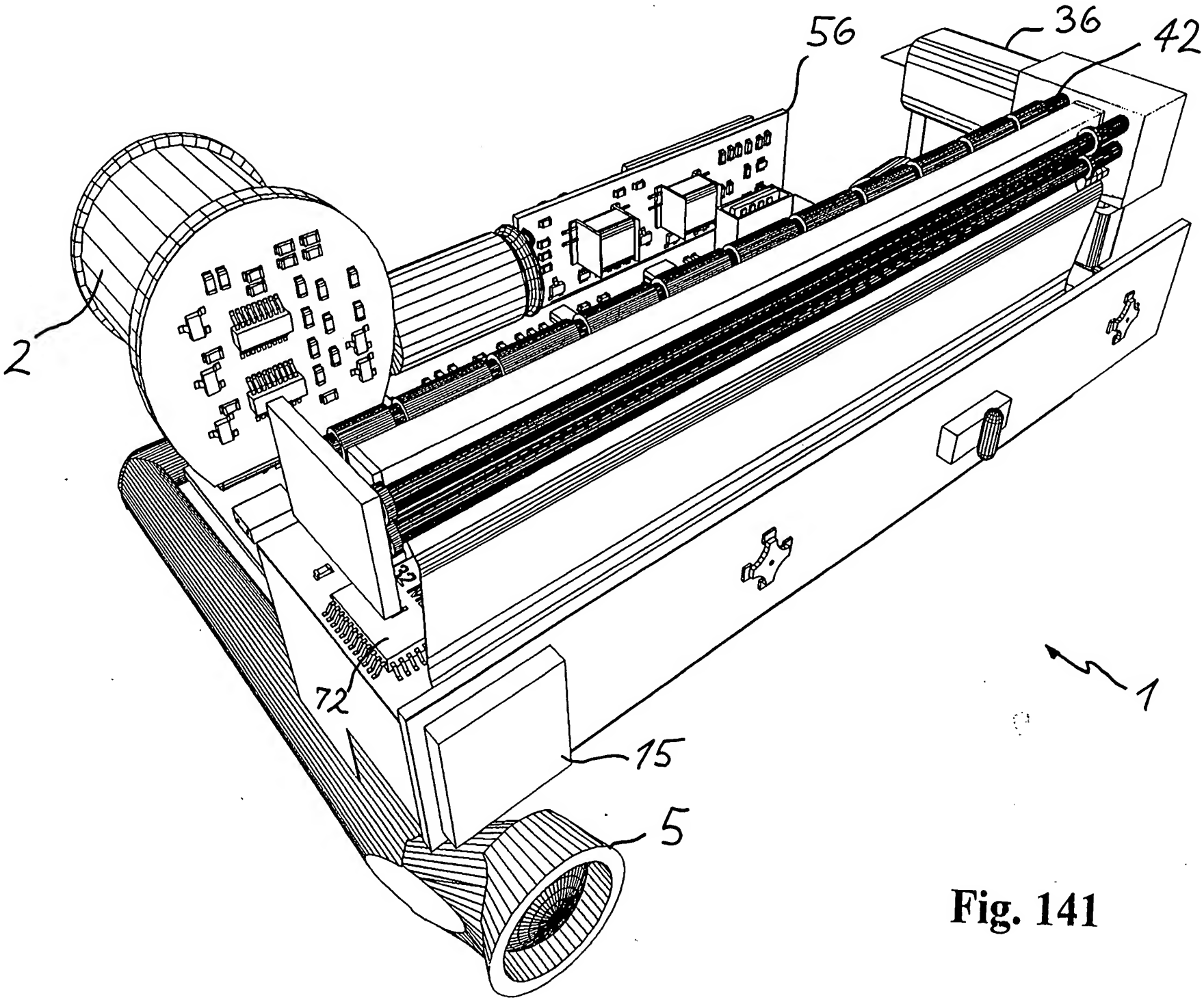



Fig. 141

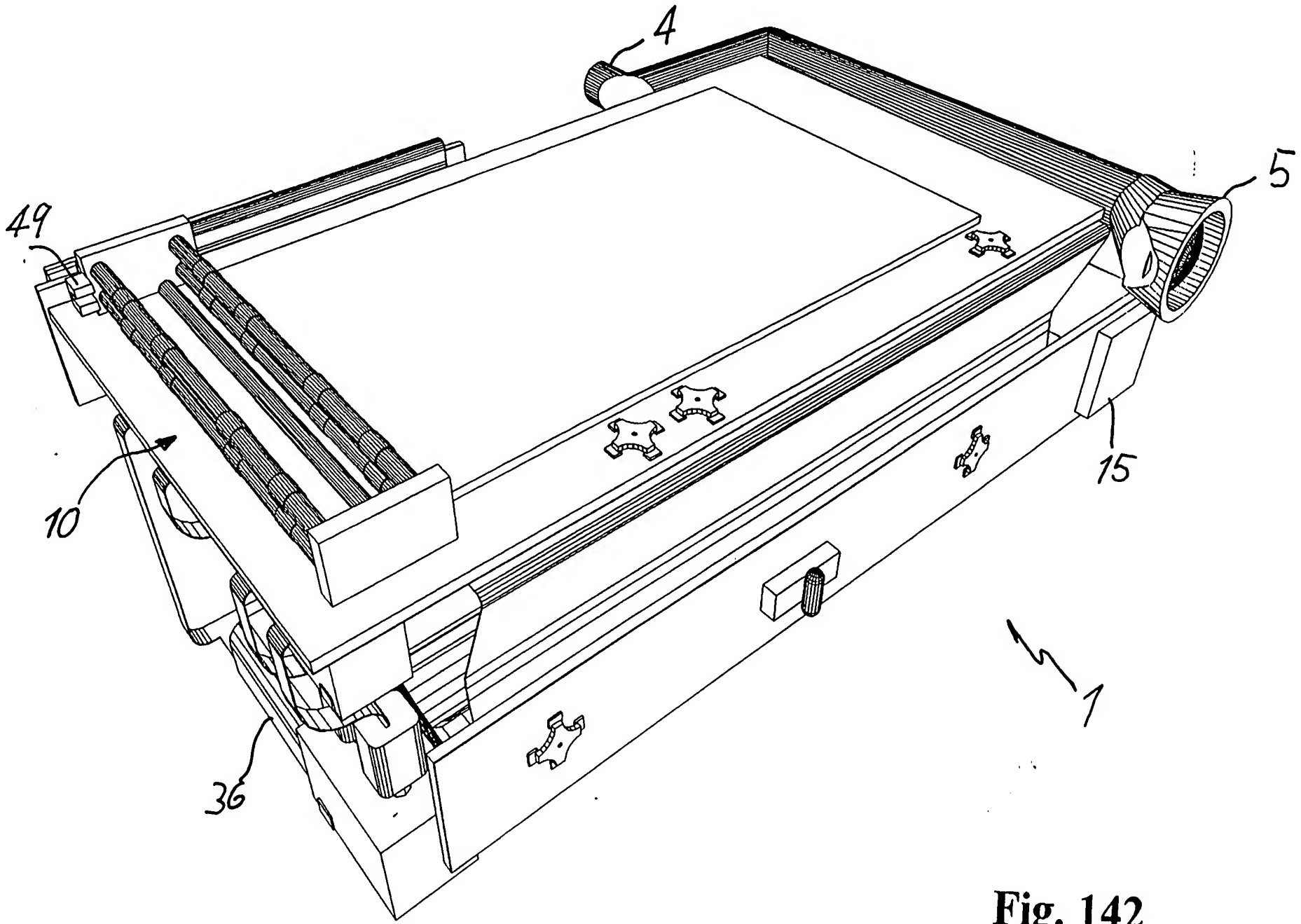



Fig. 142

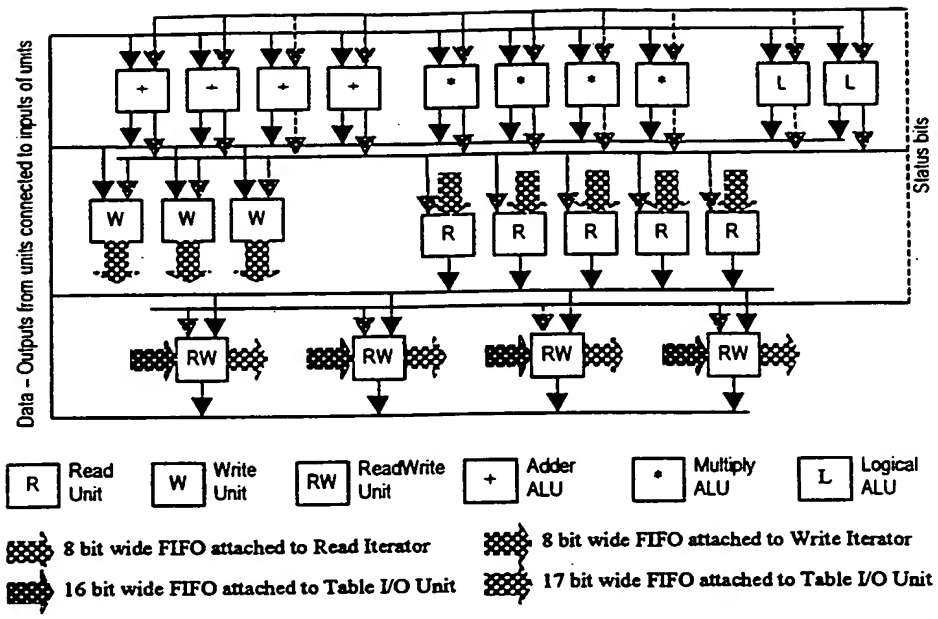



Fig. 143

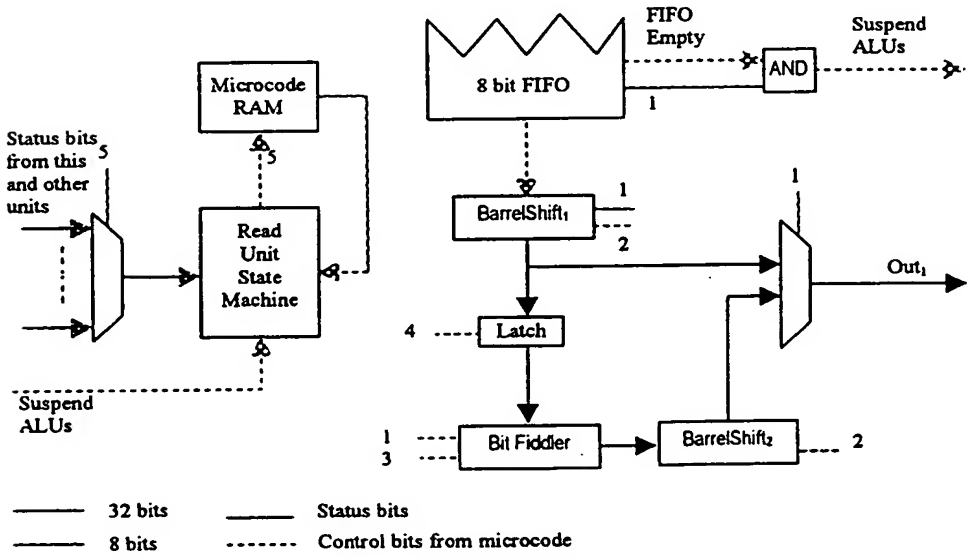
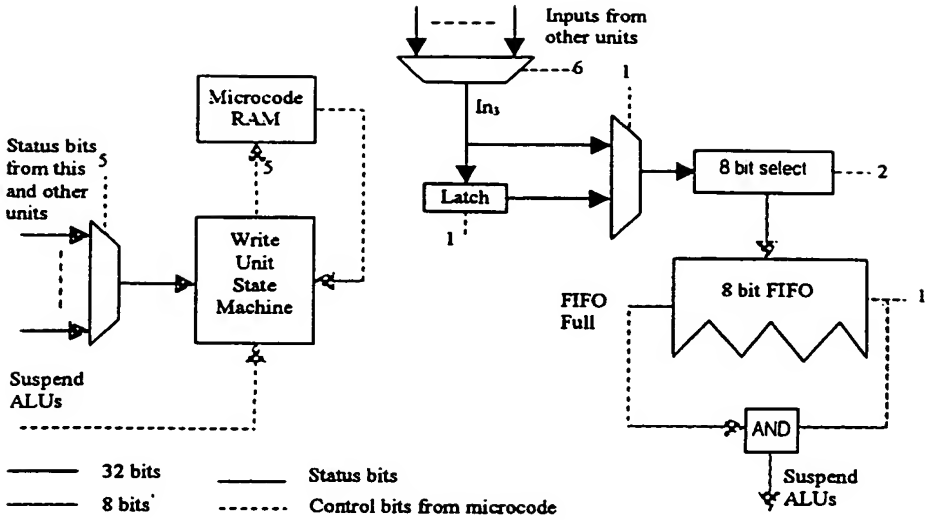



Fig. 144


Inputs from other units
6
1
In3
Microcode RAM
5
Status bits from this and other units
5
Write Unit State Machine
Latch
1
8 bit select
2
8 bit FIFO
1
FIFO Full
AND
Suspend ALUs
Suspend ALUs
32 bits
8 bits
Status bits
Control bits from microcode

# Fig. 145

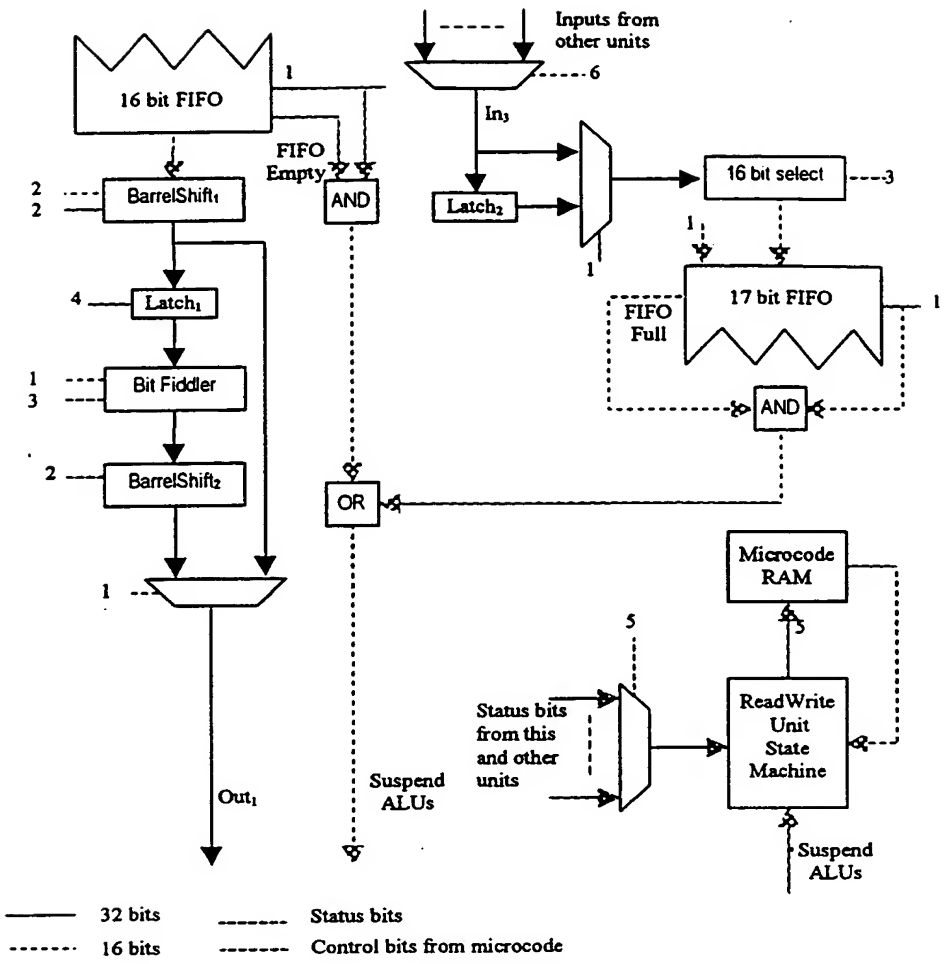
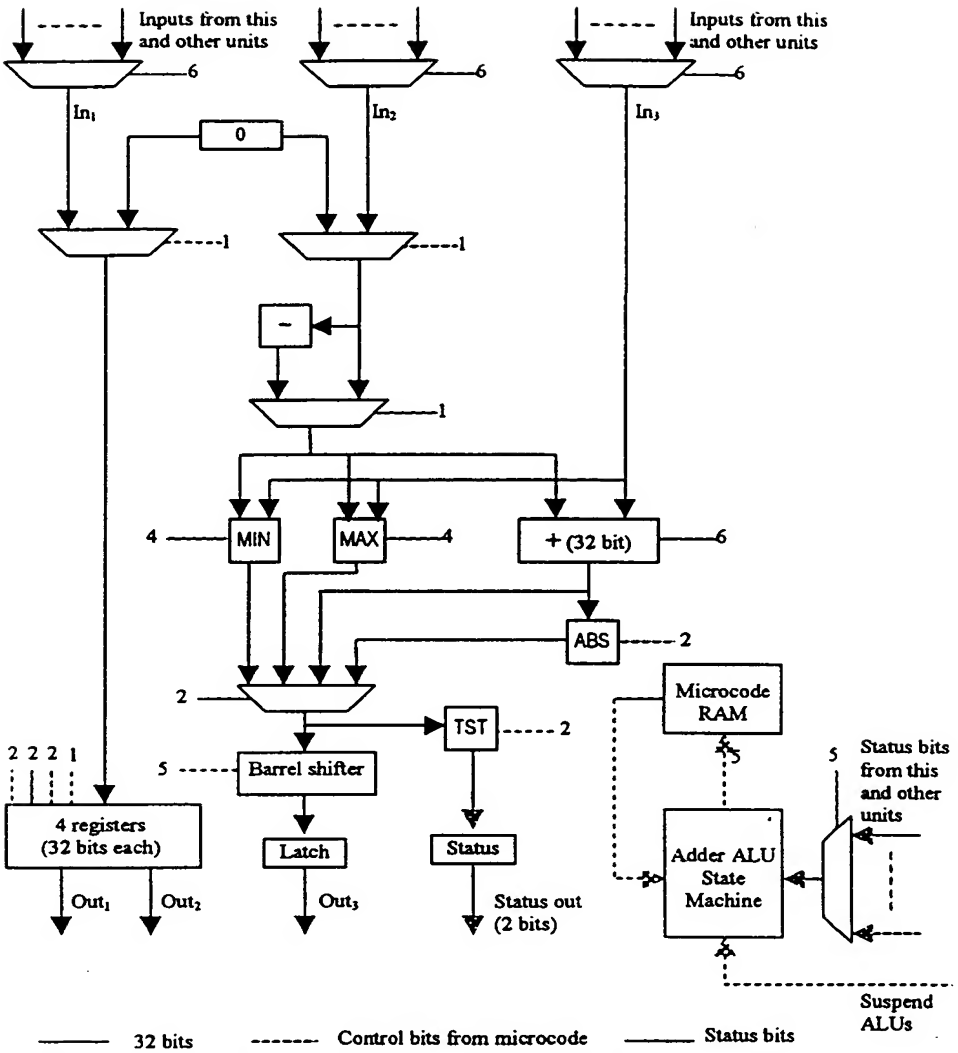

16 bit FIFO
1
FIFO Empty
AND
2
2
BarrelShift1
4
Latch1
1
3
Bit Fiddler
2
BarrelShift2
1
Out1
Inputs from other units
6
In3
Latch2
16 bit select
3
1
1
17 bit FIFO
1
FIFO Full
AND
OR
Suspend ALUs
Microcode RAM
5
5
Status bits from this and other units
ReadWrite Unit State Machine
Suspend ALUs
32 bits
16 bits
Status bits
Control bits from microcode

Fig. 146




Note: The control bits as input to MIN, MAX, +, ABS, and TST are common, taking 5 bits in total.

**Fig. 147**

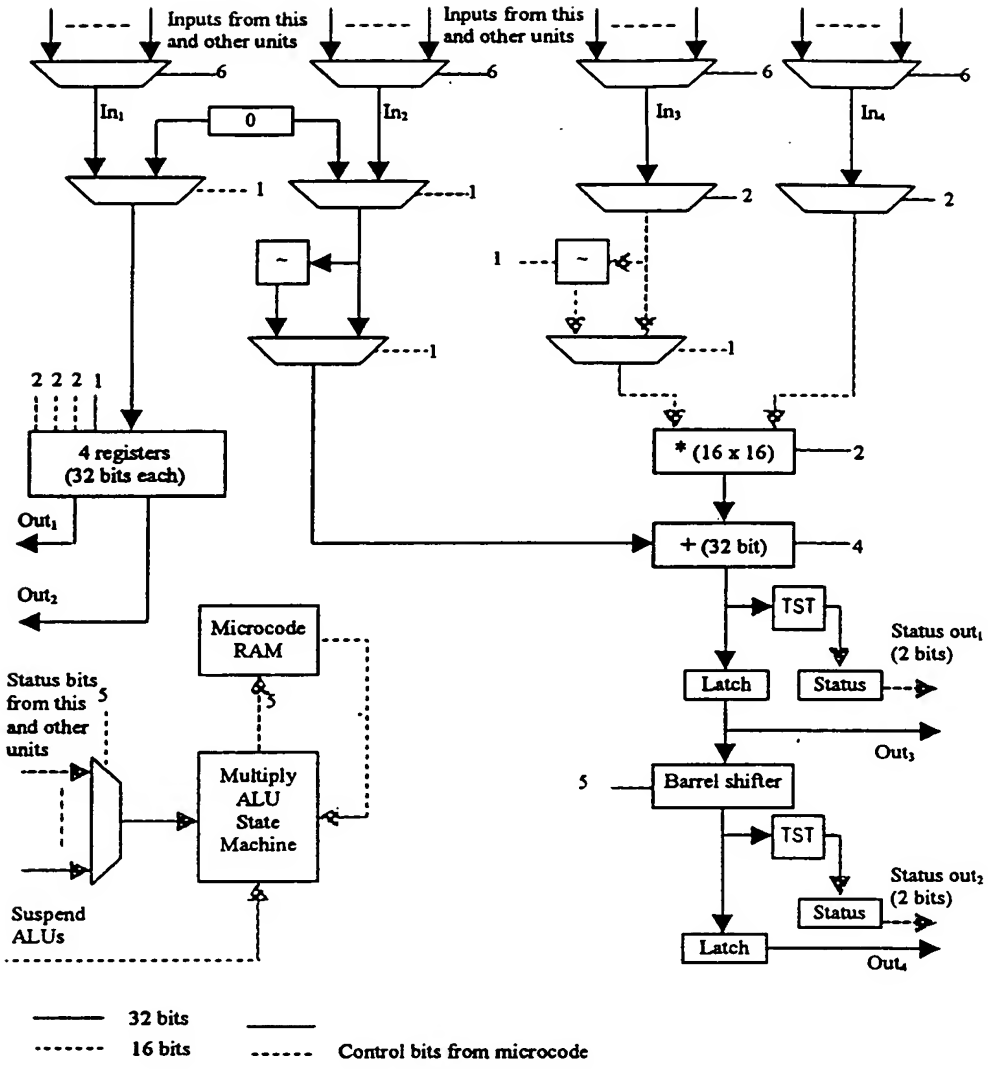

Inputs from this and other units
Inputs from this and other units
6
6
6
6
In1
0
In2
In3
In4
1
1
2
2
~
1
~
1
1
2 2 2 1
4 registers (32 bits each)
Out1
Out2
* (16 x 16)
2
+ (32 bit)
4
TST
Status out1 (2 bits)
Latch
Status
Out3
Microcode RAM
5
Status bits from this and other units
5
Multiply ALU State Machine
5
Barrel shifter
TST
Status out2 (2 bits)
Status
Latch
Out4
Suspend ALUs
32 bits
16 bits
Control bits from microcode

Fig. 148

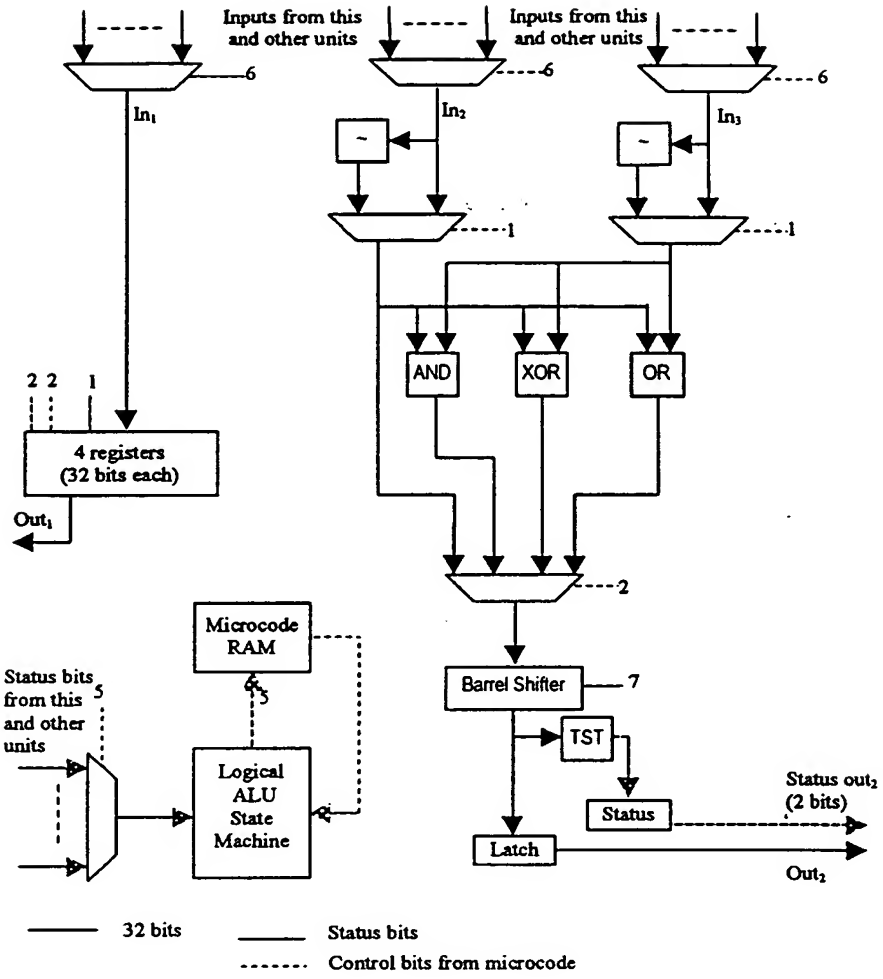

Inputs from this and other units
Inputs from this and other units
6
6
6
In1
In2
In3
~
~
1
1
AND
XOR
OR
2 2 1
4 registers (32 bits each)
Out1
2
Microcode RAM
Status bits from this and other units
5
5
Logical ALU State Machine
Barrel Shifter
7
TST
Status
Latch
Status out2 (2 bits)
Out2
32 bits
Status bits
Control bits from microcode

Fig. 149

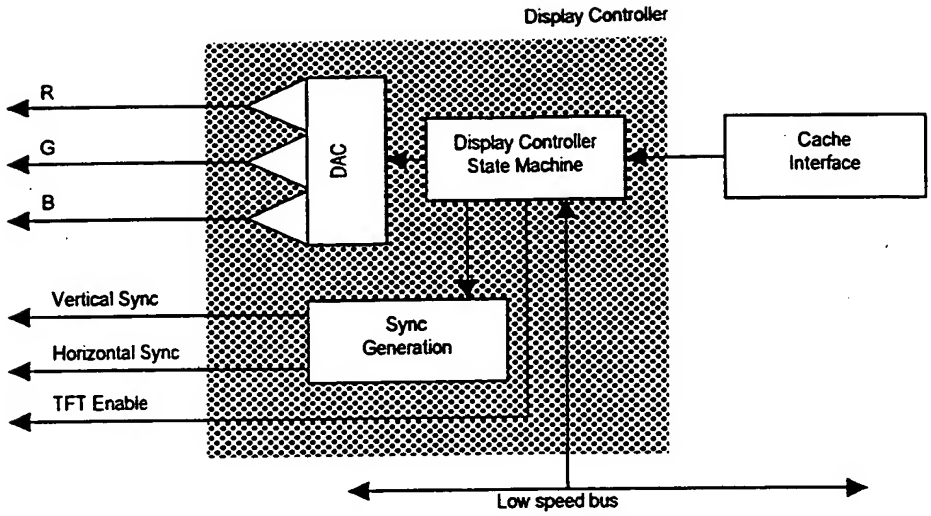

Display Controller
R
G
B
DAC
Display Controller
State Machine
Cache
Interface
Vertical Sync
Horizontal Sync
TFT Enable
Sync
Generation
Low speed bus

Fig. 150

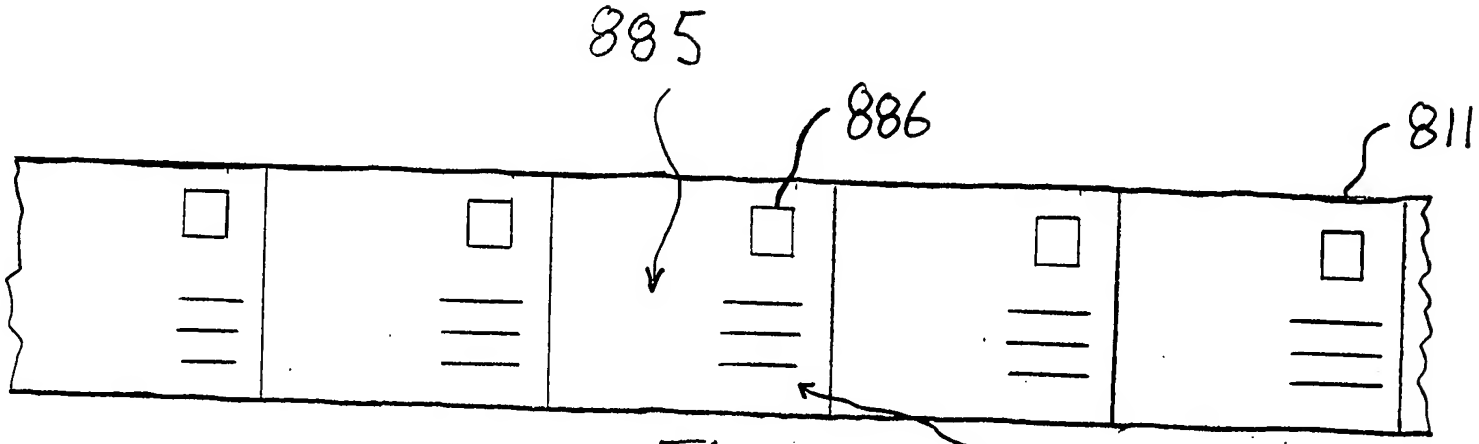



Fig. 151

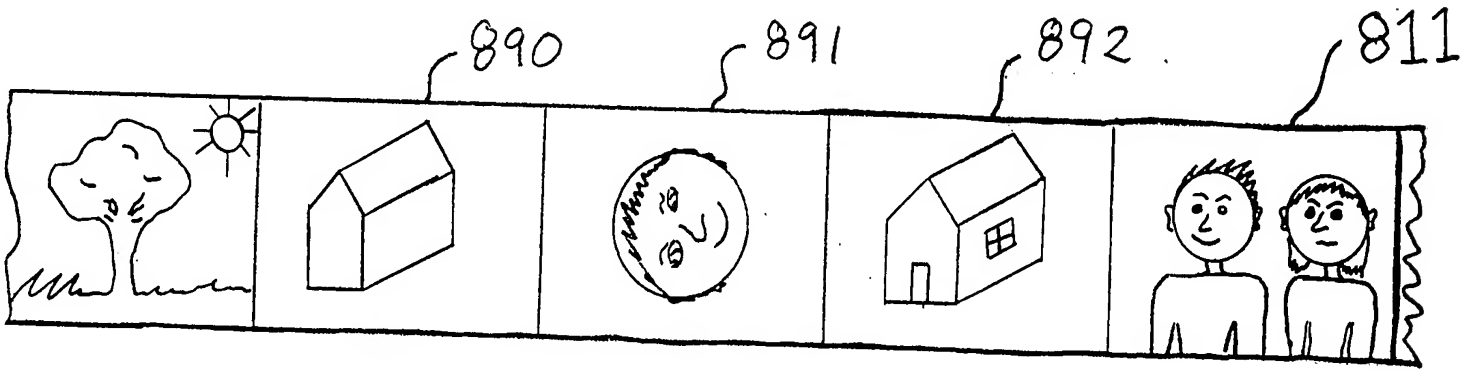



Fig. 152

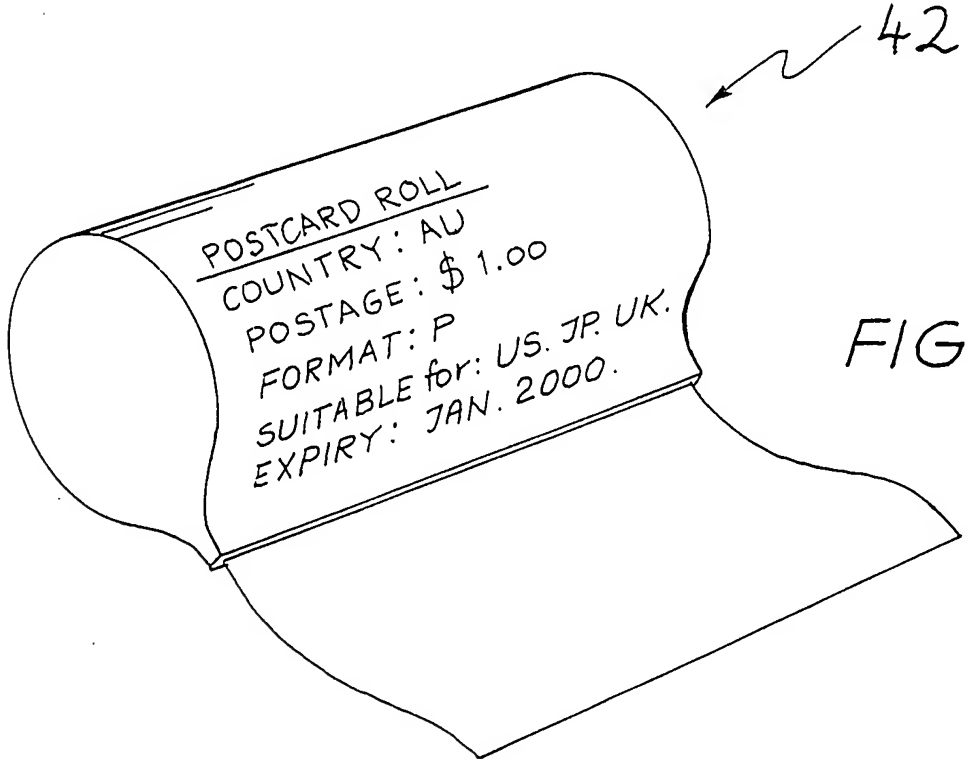

42
POSTCARD ROLL
COUNTRY: AU
POSTAGE: $ 1.00
FORMAT: P
SUITABLE for: US. JP. UK.
EXPIRY: JAN. 2000.

FIG. 153

## Appendix A – Related Australian Provisional Patent Applications

The present provisional is one of a series of interrelated Australian Provisional Patent Applications filed concurrently by the present Applicant and which together relate to a new image processing system which presents a large number of significant advances in a number of technological fields. These fields include, but are not limited to those set out in the following table:

- Camera technologies
- Display technologies
- Image processing
- Ink Jet printing technology
- Semiconductor fabrication technology
- Micro Electro Mechanical Systems (MEMS)
- VLSI and ULSI fabrication including Thin Field Techology
- Magnetics
- Fluid dynamics
- Precision engineering
- Plastics molding
- Materials science
- Digital systems architecture
- Fluid Dynamics
- Precision Engineering
- Non-impact printing technologies
- Mechanical and stress analysis
- Ink Chemistry
- Electronics
- Electrostatics

Naturally with such a large number of significant advances, it is necessary to read this Application with its associated Australian Provisional Patent Applications to gain a thorough understanding of the operation of these technologies. The following tables set out a full list of the associated Australian Provisional Patent Applications filed concurrently herewith by the present applicant which should be referred to in obtaining a full understanding of the operation of the present invention:

### Ink Jet Printing

A large number of new forms of ink jet printers have been developed to facilitate alternative ink jet technologies for the image processing system. Australian Provisional Patent Applications relating to these ink jets include:

Image Creation Method and Apparatus (IJ01)
Image Creation Method and Apparatus (IJ02)
Image Creation Method and Apparatus (IJ03)
Image Creation Method and Apparatus (IJ04)
Image Creation Method and Apparatus (IJ05)
Image Creation Method and Apparatus (IJ06)
Image Creation Method and Apparatus (IJ07)
Image Creation Method and Apparatus (IJ08)
Image Creation Method and Apparatus (IJ09)
Image Creation Method and Apparatus (IJ10)
Image Creation Method and Apparatus (IJ11)
Image Creation Method and Apparatus (IJ12)
Image Creation Method and Apparatus (IJ13)
Image Creation Method and Apparatus (IJ14)

Image Creation Method and Apparatus (IJ15)
Image Creation Method and Apparatus (IJ16)
Image Creation Method and Apparatus (IJ17)
Image Creation Method and Apparatus (IJ18)
Image Creation Method and Apparatus (IJ19)
Image Creation Method and Apparatus (IJ20)
Image Creation Method and Apparatus (IJ21)
Image Creation Method and Apparatus (IJ22)
Image Creation Method and Apparatus (IJ23)
Image Creation Method and Apparatus (IJ24)
Image Creation Method and Apparatus (IJ25)
Image Creation Method and Apparatus (IJ26)
Image Creation Method and Apparatus (IJ27)
Image Creation Method and Apparatus (IJ28)
Image Creation Method and Apparatus (IJ29)
Image Creation Method and Apparatus (IJ30)
Supply Method and Apparatus (F1)
Supply Method and Apparatus (F2)

**Ink Jet Manufacturing**

Significant developments have occurred in the field of ink jet print head construction. These advances are included in the following Australian Provisional Patent Applications.

A Method of Manufacture of an Image Creation Apparatus (IJM01)
A Method of Manufacture of an Image Creation Apparatus (IJM02)
A Method of Manufacture of an Image Creation Apparatus (IJM03)
A Method of Manufacture of an Image Creation Apparatus (IJM04)
A Method of Manufacture of an Image Creation Apparatus (IJM05)
A Method of Manufacture of an Image Creation Apparatus (IJM06)
A Method of Manufacture of an Image Creation Apparatus (IJM07)
A Method of Manufacture of an Image Creation Apparatus (IJM08)
A Method of Manufacture of an Image Creation Apparatus (IJM09)
A Method of Manufacture of an Image Creation Apparatus (IJM10)
A Method of Manufacture of an Image Creation Apparatus (IJM11)
A Method of Manufacture of an Image Creation Apparatus (IJM12)
A Method of Manufacture of an Image Creation Apparatus (IJM13)
A Method of Manufacture of an Image Creation Apparatus (IJM14)
A Method of Manufacture of an Image Creation Apparatus (IJM15)
A Method of Manufacture of an Image Creation Apparatus (IJM16)
A Method of Manufacture of an Image Creation Apparatus (IJM17)
A Method of Manufacture of an Image Creation Apparatus (IJM18)
A Method of Manufacture of an Image Creation Apparatus (IJM19)
A Method of Manufacture of an Image Creation Apparatus (IJM20)
A Method of Manufacture of an Image Creation Apparatus (IJM21)
A Method of Manufacture of an Image Creation Apparatus (IJM22)
A Method of Manufacture of an Image Creation Apparatus (IJM23)
A Method of Manufacture of an Image Creation Apparatus (IJM24)
A Method of Manufacture of an Image Creation Apparatus (IJM25)
A Method of Manufacture of an Image Creation Apparatus (IJM26)
A Method of Manufacture of an Image Creation Apparatus (IJM27)
A Method of Manufacture of an Image Creation Apparatus (IJM28)
A Method of Manufacture of an Image Creation Apparatus (IJM29)
A Method of Manufacture of an Image Creation Apparatus (IJM30)

**MEMS Technology**

The following application relate to Micro Electro-Mechanical Systems technologies:

A device (MEMS01)
A device (MEMS02)
A device (MEMS03)
A device (MEMS04)
A device (MEMS05)
A device (MEMS06)
A device (MEMS07)
A device (MEMS08)
A device (MEMS09)
A device (MEMS10)

**Artcam Technologies**

The following Australian Provisional Patent Applications relate to the a new field of image processing technology known as Artcam.

Image Processing Method and Apparatus (ART01)
Image Processing Method and Apparatus (ART02)
Image Processing Method and Apparatus (ART03)
Image Processing Method and Apparatus (ART05)
Image Processing Method and Apparatus (ART06)
Media Device (ART07)
Image Processing Method and Apparatus (ART08)
Image Processing Method and Apparatus (ART09)
Image Processing Method and Apparatus (ART10)
Image Processing Method and Apparatus (ART11)
Image Processing Method and Apparatus (ART12)
Media Device (ART13)
Image Processing Method and Apparatus (ART12)
Media Device (ART15)
Media Device (ART16)
Media Device (ART17)
Media Device (ART18)
Data Processing Method and Apparatus (ART19)
Data Processing Method and Apparatus (ART20)
Media Processing Method and Apparatus (ART21)
Image Processing Method and Apparatus (ART22)
Image Processing Method and Apparatus (ART23)
Image Processing Method and Apparatus (ART24)
Image Processing Method and Apparatus (ART25)
Image Processing Method and Apparatus (ART26)
Image Processing Method and Apparatus (ART27)
Data Processing Method and Apparatus (ART29)
Data Processing Method and Apparatus (ART32)
Image Processing Method and Apparatus (ART33)
Sensor Creation Method and Apparatus (ART36)
Data Processing Method and Apparatus (ART37)
Data Processing Method and Apparatus (ART38)
Data Processing Method and Apparatus (ART39)
Data Processing Method and Apparatus (ART40)
Data Processing Method and Apparatus (ART43)
Data Processing Method and Apparatus (ART44)
Data Processing Method and Apparatus (ART45)
Data Processing Method and Apparatus (ART46)

Data Processing Method and Apparatus (ART50)
Data Processing Method and Apparatus (ART51)
Data Processing Method and Apparatus (ART52)
Image Processing Method and Apparatus (ART53)
Image Processing Method and Apparatus (ART54)
Image Processing Method and Apparatus (ART56)

## Appendix B – Ink Jet Types

### Physical Variables

The present invention utilises an ink jet printer. In the construction of ink jet printing devices many trade offs can be made and many factors will influence the final form of any particular ink jet printer device. For an example of the many different factors utilized in the construction of ink jet printers reference is made to the annual proceedings of the IS & T International Congress on Advances in Non-Impact Printing Technologies.

The present invention, in utilizing an ink jet printer, presents a number of different alternatives in construction, many of which have been independently invented by Kia Silverbrook. As a result of these inventions, it is considered that a large number of choices could be made by the person skilled in the art as a result of this and previous disclosures in the field of construction of ink jet print heads. These independent variations can be grouped in accordance with the following table:

| Physical Variable | Number of Values |
|---|---|
| Actuator mechanism | 18 entries |
| Basic operation mode | 7 entries |
| Auxiliary mechanism | 8 entries |
| Actuator amplification or modification method | 13 entries |
| Actuator motion | 14 entries |
| Nozzle refill method | 4 entries |
| Nozzle plate construction | 8 entries |
| Drop ejection direction | 5 entries |
| Ink type | 6 entries |

Obviously, selection of suitable values from the above table will result in a plethora of different possible designs of which a large number will be commercially viable. A number of these designs have been invented by other parties and many have been indicated in the table below. In addition, Kia Silverbrook has previously invented one form of ink jet printer, hereinafter known as LIFT and disclosed in a series of applications filed as Australian Provisional Patent Applications on 12 April, 1995 including application PN2308 entitled "A Liquid Ink Fault Tolerant Ink Mechanism".

The accompanying appendix C sets out a selection of the main ink jet configurations involving novel arrangements of the above variables. These ink jet configurations have been denoted IJ01 to IJ30.

For further discussion of the operation of each of the ink jets IJ01 to IJ30, reference is made to the corresponding Australian Provisional Patent Applications filed concurrently herewith. Appendix A, attached to the present specification, contains a list of all the Australian Provisional Patent Applications filed concurrently herewith as part of this project. Each of the ink jet patents are denoted with the "IJ" number in their title. Each of the IJ01 to IJ30 examples can be made into ink jet print heads with superior characteristics to any currently available ink jet technology.

### Variations in physical variables

The following tables set out a number of variations in each physical variable in the table listed above. These tables only address drop-on-demand ink jet technologies. Continuous ink jet technologies are not specifically addressed.

Actuator mechanism (applied only to selected nozzles)

| Actuator Mechanism | Description | Advantages | Disadvantages | Examples |
|---|---|---|---|---|
| **Thermal bubble** | An electrothermal heater heats the ink to above boiling point, transferring significant heat to the aqueous ink. A bubble nucleates and quickly forms, expelling the ink.<br>The efficiency of the process is low, with typically less than 0.05% of the electrical energy being transformed into kinetic energy of the drop. | ◆ High force generated<br>◆ Simple construction<br>◆ No moving parts<br>◆ Fast operation<br>◆ Small chip area required for actuator | ◆ High power<br>◆ Ink carrier limited to water<br>◆ Low efficiency<br>◆ High temperatures required<br>◆ High mechanical stress<br>◆ Unusual materials required<br>◆ Large drive transistors<br>◆ Cavitation causes actuator failure<br>◆ Kogation reduces bubble formation<br>◆ Large print heads are difficult to fabricate | ◆ Canon Bubblejet 1979 Endo et al GB patent 2,007,162<br>◆ Xerox heater-in-pit 1990 Hawkins et al USP 4,899,181<br>◆ Hewlett-Packard TIJ 1982 Vaught et al USP 4,490,728 |
| **Piezoelectric** | A piezoelectric crystal such as lead lanthanum zirconate (PZT) is electrically activated, and either expands, shears, or bends to apply pressure to the ink, ejecting drops. | ◆ Low power consumption<br>◆ Many ink types can be used<br>◆ Fast operation<br>◆ High efficiency | ◆ Very large area required for actuator<br>◆ Difficult to integrate with electronics<br>◆ High voltage drive transistors required<br>◆ Full pagewidth print heads impractical due to actuator size<br>◆ Requires electrical poling in high field strengths during manufacture | ◆ Kyser et al USP 3,946,398<br>◆ Zoltan USP 3,683,212<br>◆ 1973 Stemme USP 3,747,120<br>◆ Epson Stylus<br>◆ Tektronix<br>◆ IJ04 |
| **Electro-strictive** | An electric field is used to activate electrostriction in relaxor materials such as lead lanthanum zirconate titanate (PLZT) or lead magnesium niobate (PMN). | ◆ Low power consumption<br>◆ Many ink types can be used<br>◆ Low thermal expansion<br>◆ Electric field strength required (approx. 3.5 V/μm) can be generated without difficulty<br>◆ Does not require electrical poling | ◆ Low maximum strain (approx. 0.01%)<br>◆ Large area required for actuator due to low strain<br>◆ Response speed is marginal (~ 10 μs)<br>◆ High voltage drive transistors required<br>◆ Full pagewidth print heads impractical due to actuator size | ◆ IJ04 |
| **Ferroelectric** | An electric field is used to induce a phase transition between the antiferroelectric (AFE) and ferroelectric (FE) phase. Perovskite materials such as tin modified lead lanthanum zirconate titanate (PLZSnT) exhibit large strains of up to 1% associated with the AFE to FE phase transition. | ◆ Low power consumption<br>◆ Many ink types can be used<br>◆ Fast operation (< 1 μs)<br>◆ Relatively high longitudinal strain<br>◆ High efficiency<br>◆ Electric field strength of around 3 V/μm can be readily provided | ◆ Difficult to integrate with electronics<br>◆ Unusual materials such as PLZSnT are required<br>◆ Actuators require a large area | ◆ IJ04 |

| | | | | |
|---|---|---|---|---|
| **Electrostatic plates** | Conductive plates are separated by a compressible or fluid dielectric (usually air). Upon application of a voltage, the plates attract each other and displace ink, causing drop ejection. The conductive plates may be in a comb or honeycomb structure, or stacked to increase the surface area and therefore the force. | ♦ Low power consumption<br>♦ Many ink types can be used<br>♦ Fast operation | ♦ Difficult to operate electrostatic devices in an aqueous environment<br>♦ The electrostatic actuator will normally need to be separated from the ink<br>♦ Very large area required to achieve high forces<br>♦ High voltage drive transistors may be required<br>♦ Full pagewidth print heads are not competitive due to actuator size | ♦ IJ02, IJ04 |
| **Electrostatic pull on ink** | A strong electric field is applied to the ink, whereupon electrostatic attraction accelerates the ink towards the print medium. | ♦ Low current consumption<br>♦ Low temperature | ♦ High voltage required<br>♦ May be damaged by sparks due to air breakdown<br>♦ Required field strength increases as the drop size decreases<br>♦ High voltage drive transistors required<br>♦ Electrostatic field attracts dust | ♦ 1989 Saito et al, USP 4,799,068<br>♦ 1989 Miura et al, USP 4,810,954 |
| **Permanent magnet electro-magnetic** | An electromagnet directly attracts a permanent magnet, displacing ink and causing drop ejection. Rare earth magnets with a field strength around 1 Tesla can be used. Examples are: Samarium Cobalt (SaCo) and magnetic materials in the neodymium iron boron family (NdFeB, NdDyFeBNb, NdDyFeB, etc) | ♦ Low power consumption<br>♦ Many ink types can be used<br>♦ Fast operation<br>♦ High efficiency<br>♦ Easy extension from single nozzles to pagewidth print heads | ♦ Complex fabrication<br>♦ Permanent magnetic material such as Neodymium Iron Boron (NdFeB) required.<br>♦ High local currents required<br>♦ Copper metallization should be used for long electromigration lifetime and low resistivity<br>♦ Pigmented inks are usually infeasible<br>♦ Operating temperature limited to the Curie temperature (around 540 K) | ♦ IJ07, IJ10 |
| **Soft magnetic core electro-magnetic** | A solenoid induced a magnetic field in a soft magnetic core or yoke fabricated from a ferrous material such as nickel iron (NiFe). Typically, the soft magnetic material is in two parts, which are normally held apart by a spring. When the solenoid is actuated, the two parts attract, displacing the ink. | ♦ Low power consumption<br>♦ Many ink types can be used<br>♦ Fast operation<br>♦ High efficiency<br>♦ Easy extension from single nozzles to pagewidth print heads | ♦ Complex fabrication<br>♦ Unusual materials such as nickel iron (NiFe) are required<br>♦ High local currents required<br>♦ Copper metallization should be used for long electromigration lifetime and low resistivity<br>♦ Pigmented inks are usually infeasible | ♦ IJ01, IJ05, IJ08, IJ10<br>♦ IJ12, IJ14, IJ15, IJ17 |

| | | | | |
|---|---|---|---|---|
| **Magnetic Lorenz force** | The Lorenz force acting on a current carrying wire in a magnetic field is utilized.<br>This allows the magnetic field to be supplied externally to the print head, for example with rare earth permanent magnets.<br>Only the current carrying wire need be fabricated on the print-head, simplifying materials requirements. | ♦ Low power consumption<br>♦ Many ink types can be used<br>♦ Fast operation<br>♦ High efficiency<br>♦ Easy extension from single nozzles to pagewidth print heads | ♦ Force acts as a twisting motion<br>♦ Typically, only a quarter of the solenoid length provides force in a useful direction<br>♦ High local currents required<br>♦ Copper metallization should be used for long electromigration lifetime and low resistivity<br>♦ Pigmented inks are usually infeasible | ♦ IJ06, IJ11, IJ13, IJ16 |
| **Magneto-striction** | The actuator uses the giant magnetostrictive effect of materials such as Terfenol-D (an alloy of terbium, dysprosium and iron developed at the Naval Ordnance Laboratory, hence Ter-Fe-NOL). For best efficiency, the actuator should be pre-stressed to approx. 8 MPa. | ♦ Many ink types can be used<br>♦ Fast operation<br>♦ Easy extension from single nozzles to pagewidth print heads<br>♦ High force is available | ♦ Force acts as a twisting motion<br>♦ Unusual materials such as Terfenol-D are required<br>♦ High local currents required<br>♦ Copper metallization should be used for long electromigration lifetime and low resistivity<br>♦ Pre-stressing may be required | ♦ IJ25 |
| **Surface tension reduction** | Ink under positive pressure is held in a nozzle by surface tension. The surface tension of the ink is reduced below the bubble threshold, causing the ink to egress from the nozzle. | ♦ Low power consumption<br>♦ Simple construction<br>♦ No unusual materials required in fabrication<br>♦ High efficiency<br>♦ Easy extension from single nozzles to pagewidth print heads | ♦ Requires supplementary force to effect drop separation<br>♦ Requires special ink surfactants<br>♦ Speed may be limited by surfactant properties | ♦ LIFT |
| **Viscosity reduction** | The ink viscosity is locally reduced to select which drops are to be ejected. A viscosity reduction can be achieved electrothermally with most inks, but special inks can be engineered for a 100:1 viscosity reduction. | ♦ Simple construction<br>♦ No unusual materials required in fabrication<br>♦ Easy extension from single nozzles to pagewidth print heads | ♦ Requires supplementary force to effect drop separation<br>♦ Requires special ink viscosity properties<br>♦ High speed is difficult to achieve<br>♦ Requires oscillating ink pressure<br>♦ A high temperature difference (typically 80 degrees) is required | ♦ LIFT |
| **Acoustic** | An acoustic wave is generated and focussed upon the drop ejection region. | ♦ Can operate without a nozzle plate | ♦ Complex drive circuitry<br>♦ Complex fabrication<br>♦ Low efficiency<br>♦ Poor control of drop position<br>♦ Poor control of drop volume | ♦ 1993 Hadimioglu et al, EUP 550,192<br>♦ 1993 Elrod et al, EUP 572,220 |

| | | | | |
|---|---|---|---|---|
| **Thermoelastic bend actuator** | An actuator which relies upon differential thermal expansion upon Joule heating is used. | ◆ Low power consumption<br>◆ Many ink types can be used<br>◆ Simple planar fabrication<br>◆ Small chip area required for each actuator<br>◆ Fast operation<br>◆ High efficiency<br>◆ CMOS compatible voltages and currents<br>◆ Standard MEMS processes can be used<br>◆ Easy extension from single nozzles to pagewidth print heads | ◆ Difficult to achieve a large force, large travel, and small size simultaneously<br>◆ Efficient aqueous operation requires a thermal insulator on the hot arm<br>◆ Corrosion prevention can be difficult<br>◆ Pigmented inks may be infeasible, as pigment particles may jam the bend actuator | ◆ IJ03, IJ09, IJ17, IJ18<br>◆ IJ19, IJ20, IJ21, IJ22<br>◆ IJ23, IJ24, IJ27, IJ28<br>◆ IJ29, IJ30 |
| **High CTE thermoelastic actuator** | A material with a very high coefficient of thermal expansion (CTE) such as polytetrafluoroethylene (PTFE) is used. As high CTE materials are usually non conductive, a heater fabricated from a conductive material is incorporated. A 50 µm long PTFE bend actuator with polysilicon heater and 15 mW power input can provide 180 µN force and 10 µm deflection. Actuator motions include:<br>1) Bend<br>2) Push<br>3) Buckle<br>4) Rotate | ◆ High force can be generated<br>◆ Three methods of PTFE deposition are under development: chemical vapor deposition (CVD), spin coating, and evaporation<br>◆ PTFE is a candidate for low dielectric constant insulation in ULSI<br>◆ Very low power consumption<br>◆ Many ink types can be used<br>◆ Simple planar fabrication<br>◆ Small chip area required for each actuator<br>◆ Fast operation<br>◆ High efficiency<br>◆ CMOS compatible voltages and currents<br>◆ Easy extension from single nozzles to pagewidth print heads | ◆ Requires special material (e.g. PTFE)<br>◆ Requires a PTFE deposition process, which is not yet standard in ULSI fabs<br>◆ PTFE deposition cannot be followed with high temperature (above 350 °C) processing<br>◆ Pigmented inks may be infeasible, as pigment particles may jam the bend actuator | ◆ IJ09, IJ17, IJ18, IJ20<br>◆ IJ21, IJ22, IJ23, IJ24<br>◆ IJ27, IJ28, IJ29, IJ30 |

| | | | | |
|---|---|---|---|---|
| **Conductive polymer thermoelastic actuator** | A polymer with a high coefficient of thermal expansion (such as PTFE) is doped with conducting substances to increase its conductivity to about 3 orders of magnitude below that of copper. The conducting polymer expands when resistively heated.<br>Examples of conducting dopants include:<br>1) Carbon nanotubes<br>2) Metal fibers<br>3) Conductive polymers such as doped polythiophene<br>4) Carbon granules | ♦ High force can be generated<br>♦ Very low power consumption<br>♦ Many ink types can be used<br>♦ Simple planar fabrication<br>♦ Small chip area required for each actuator<br>♦ Fast operation<br>♦ High efficiency<br>♦ CMOS compatible voltages and currents<br>♦ Easy extension from single nozzles to pagewidth print heads | ♦ Requires special materials development (High CTE conductive polymer)<br>♦ Requires a PTFE deposition process, which is not yet standard in ULSI fabs<br>♦ PTFE deposition cannot be followed with high temperature (above 350 °C) processing<br>♦ Evaporation and CVD deposition techniques cannot be used<br>♦ Pigmented inks may be infeasible, as pigment particles may jam the bend actuator | ♦ IJ24 |
| **Shape memory alloy** | A shape memory alloy such as TiNi (also known as Nitinol - Nickel Titanium alloy developed at the Naval Ordnance Laboratory) is thermally switched between its weak martensitic state and its high stiffness austenic state. The shape of the actuator in its martensitic state is deformed relative to the austenic shape. The shape change causes ejection of a drop. | ♦ High force is available (stresses of hundreds of MPa)<br>♦ Large strain is available (more than 3%)<br>♦ High corrosion resistance<br>♦ Simple construction<br>♦ Easy extension from single nozzles to pagewidth print heads<br>♦ Low voltage operation | ♦ Fatigue limits maximum number of cycles<br>♦ Low strain (1%) is required to extend fatigue resistance<br>♦ Cycle rate limited by heat removal<br>♦ Requires unusual materials (TiNi)<br>♦ The latent heat of transformation must be provided<br>♦ High current operation<br>♦ Requires pre-stressing to distort the martensitic state | ♦ IJ26 |
| **Linear Magnetic Actuator** | Linear magnetic actuators include the Linear Induction Actuator (LIA), Linear Permanent Magnet Synchronous Actuator (LPMSA), Linear Reluctance Synchronous Actuator (LRSA), Linear Switched Reluctance Actuator (LSRA), and the Linear Stepper Actuator (LSA). | ♦ Linear Magnetic actuators can be constructed with high thrust, long travel, and high efficiency using planar semiconductor fabrication techniques<br>♦ Long actuator travel is available<br>♦ Medium force is available<br>♦ Low voltage operation | ♦ Requires unusual semiconductor materials such as soft magnetic (NiFe)<br>♦ Some varieties also require permanent magnetic materials such as Neodymium iron boron (NdFeB)<br>♦ Requires complex multi-phase drive circuitry<br>♦ High current operation | ♦ IJ12 |

## Basic operation mode

| Operational mode | Description | Advantages | Disadvantages | Examples |
|---|---|---|---|---|
| **Actuator directly pushes ink** | This is the simplest mode of operation: the actuator directly supplies sufficient kinetic energy to expel the drop. The drop must have a sufficient velocity to overcome the surface tension. | ♦ Simple operation<br>♦ No external fields required<br>♦ Satellite drops can be avoided if drop velocity is less than 4 m/s<br>♦ Can be efficient, depending upon the actuator used | ♦ Drop repetition rate is usually limited to less than 10 KHz. However, this is not fundamental to the method, but is related to the refill method normally used<br>♦ All of the drop kinetic energy must be provided by the actuator<br>♦ Satellite drops usually form if drop velocity is greater than 4 m/s | ♦ Thermal ink jet<br>♦ Piezo-electric ink jet<br>♦ IJ01, IJ02, IJ03, IJ04<br>♦ IJ05, IJ06, IJ07, IJ09<br>♦ IJ11, IJ12, IJ14, IJ16<br>♦ IJ20, IJ22, IJ23, IJ24<br>♦ IJ25, IJ26, IJ27, IJ28<br>♦ IJ29, IJ30 |
| **Proximity** | The drops to be printed are selected by some manner (e.g. thermally induced surface tension reduction of pressurized ink). Selected drops are separated from the ink in the nozzle by contact with the print medium or a transfer roller. | ♦ Very simple print head fabrication can be used<br>♦ The drop selection means does not need to provide the energy required to separate the drop from the nozzle | ♦ Requires close proximity between the print head and the print media or transfer roller<br>♦ May require two print heads printing alternate rows of the image<br>♦ Monolithic color print heads are difficult | ♦ LIFT |
| **Electrostatic pull on ink** | The drops to be printed are selected by some manner (e.g. thermally induced surface tension reduction of pressurized ink). Selected drops are separated from the ink in the nozzle by a strong electric field. | ♦ Very simple print head fabrication can be used<br>♦ The drop selection means does not need to provide the energy required to separate the drop from the nozzle | ♦ Requires very high electrostatic field<br>♦ Electrostatic field for small nozzle sizes is above air breakdown<br>♦ Electrostatic field may attract dust | ♦ LIFT |
| **Magnetic pull on ink** | The drops to be printed are selected by some manner (e.g. thermally induced surface tension reduction of pressurized ink). Selected drops are separated from the ink in the nozzle by a strong magnetic field acting on the magnetic ink. | ♦ Very simple print head fabrication can be used<br>♦ The drop selection means does not need to provide the energy required to separate the drop from the nozzle | ♦ Requires magnetic ink<br>♦ Ink colors other than black are difficult<br>♦ Requires very high magnetic fields | ♦ LIFT |
| **Shutter** | The actuator moves a shutter to block ink flow to the nozzle. The ink pressure is pulsed at a multiple of the drop ejection frequency. | ♦ High speed (>50 KHz) operation can be achieved due to reduced refill time<br>♦ Drop timing can be very accurate<br>♦ The actuator energy can be very low | ♦ Moving parts are required<br>♦ Requires ink pressure modulator<br>♦ Friction and wear must be considered<br>♦ Stiction is possible | ♦ IJ13, IJ17, IJ21 |
| **Shuttered grill** | The actuator moves a shutter to block ink flow through a grill to the nozzle. The shutter movement need only be equal to the width of the grill holes. | ♦ Actuators with small travel can be used<br>♦ Actuators with small force can be used<br>♦ High speed (>50 KHz) operation can be achieved | ♦ Moving parts are required<br>♦ Requires ink pressure modulator<br>♦ Friction and wear must be considered<br>♦ Stiction is possible | ♦ IJ08, IJ15, IJ18, IJ19 |

| | | | | |
|---|---|---|---|---|
| **Pulsed magnetic pull on ink pusher** | A pulsed magnetic field attracts an 'ink pusher' at the drop ejection frequency. An actuator controls a catch, which prevents the ink pusher from moving when a drop is not to be ejected. | ◆ Extremely low energy operation is possible<br>◆ No heat dissipation problems | ◆ Requires an external pulsed magnetic field<br>◆ Requires special materials for both the actuator and the ink pusher<br>◆ Complex construction | ◆ IJ10 |

## Auxiliary mechanism (applied to all nozzles)

| Auxiliary Mechanism | Description | Advantages | Disadvantages | Examples |
|---|---|---|---|---|
| None | The actuator directly fires the ink drop, and there is no external field or other mechanism required. | ♦ Simplicity of construction<br>♦ Simplicity of operation<br>♦ Small physical size | ♦ Drop ejection energy must be supplied by individual nozzle actuator | ♦ Most ink jets, including piezoelectric and thermal bubble.<br>♦ IJ01, IJ02, IJ03, IJ04<br>♦ IJ05, IJ07, IJ09, IJ11<br>♦ IJ12, IJ14, IJ20, IJ22<br>♦ IJ23, IJ24, IJ25, IJ26<br>♦ IJ27, IJ28, IJ29, IJ30 |
| Oscillating ink pressure | The ink pressure oscillates, providing much of the drop ejection energy. The actuator selects which drops are to be fired by selectively blocking or enabling nozzles. The ink pressure oscillation may be achieved by vibrating the print head, or preferably by an actuator in the ink supply. | ♦ Oscillating ink pressure can provide a refill pulse, allowing higher operating speed<br>♦ The actuators may operate with much lower energy | ♦ Requires external ink pressure oscillator<br>♦ Ink pressure phase and amplitude must be carefully controlled | ♦ Viscous LIFT<br>♦ IJ08, IJ13, IJ15, IJ17<br>♦ IJ18, IJ19, IJ21 |
| Media proximity | The print head is placed in close proximity to the print medium. Selected drops protrude from the print head further than unselected drops, and contact the print medium. The drop soaks into the medium fast enough to cause drop separation. | ♦ Low power<br>♦ High accuracy<br>♦ Simple print head construction | ♦ Precision assembly required<br>♦ Paper fibers may cause problems<br>♦ Cannot print on rough substrates | ♦ Proximity LIFT |
| Transfer roller | Drops are printed to a transfer roller instead of straight to the print medium. A transfer roller can also be used for proximity drop separation. | ♦ High accuracy<br>♦ Wide range of print substrates can be used<br>♦ Ink can be dried on the transfer roller | ♦ Bulky<br>♦ Expensive<br>♦ Complex construction | ♦ Proximity LIFT<br>♦ Tektronix hot melt piezo-electric ink jet<br>♦ Any of the IJ series |
| Electrostatic | An electric field is used to accelerate selected drops towards the print medium. | ♦ Low power<br>♦ Simple print head construction | ♦ Field strength required for separation of small drops is near or above air breakdown | ♦ LIFT |
| Direct magnetic field | An magnetic field is used to accelerate selected drops of magnetic ink towards the print medium. | ♦ Low power<br>♦ Simple print head construction | ♦ Requires magnetic ink<br>♦ Requires strong magnetic field | ♦ LIFT |
| Cross magnetic field | The print head is placed in a constant magnetic field. The Lorenz force in a current carrying wire is used to move the actuator. | ♦ Does not require magnetic materials to be integrated in the print head manufacturing process | ♦ Requires external magnet<br>♦ Current densities may be high, resulting in electromigration problems | ♦ IJ06, IJ16 |
| Pulsed magnetic field | A pulsed magnetic field is used to cyclically attract a paddle which pushes on the ink. A small actuator moves a catch which selectively prevents the paddle from moving. | ♦ Very low power operation is possible<br>♦ Small print head size | ♦ Complex print head construction<br>♦ Magnetic materials required in print head | ♦ IJ10 |

## Actuator amplification or modification method

| Actuator amplification | Description | Advantages | Disadvantages | Examples |
|---|---|---|---|---|
| None | No actuator mechanical amplification is used. The actuator directly drives the drop ejection process. | ♦ Operational simplicity | ♦ Many actuator mechanisms have insufficient travel, or insufficient force, to efficiently drive the drop ejection process | ♦ Thermal Bubble Ink Jet<br>♦ IJ01, IJ02, IJ06, IJ07<br>♦ IJ16, IJ25, IJ26 |
| Differential expansion bend actuator | An actuator material expands more on one side than on the other. The expansion may be thermal, piezoelectric, magnetostrictive, or other mechanism. The bend actuator converts a high force low travel actuator mechanism to high travel, lower force mechanism. | ♦ Provides greater travel in a reduced print head area | ♦ High stresses are involved<br>♦ Care must be taken that the materials do not delaminate | ♦ Piezoelectric<br>♦ IJ03, IJ09, IJ17, IJ18<br>♦ IJ19, IJ20, IJ21, IJ22<br>♦ IJ23, IJ24, IJ27, IJ29<br>♦ IJ30 |
| Reverse spring | The actuator loads a spring. When the actuator is turned off, the spring releases. This can reverse the force/distance curve of the actuator to make it compatible with the force/time requirements of the drop ejection. | ♦ Better coupling to the ink | ♦ Fabrication complexity<br>♦ High stress in the spring | ♦ IJ05, IJ11 |
| Actuator stack | A series of thin actuators are stacked. This can be appropriate where actuators require a high electric field strength, such as electrostatic and piezoelectric actuators. | ♦ Increased travel<br>♦ Reduced drive voltage | ♦ Increased fabrication complexity<br>♦ Increased possibility of short circuits due to pinholes | ♦ IJ04 |
| Multiple actuators | Multiple smaller actuators are used simultaneously to move the ink. Each actuator need provide only a portion of the force required. | ♦ Increases the force available from an actuator<br>♦ Multiple actuators can be positioned to control ink flow accurately | ♦ Actuator forces may not add linearly, reducing efficiency | ♦ IJ12, IJ13, IJ18, IJ20<br>♦ IJ22, IJ28 |
| Linear Spring | A linear spring is used to transform a motion with small travel and high force into a longer travel, lower force motion. | ♦ Matches low travel actuator with higher travel requirements<br>♦ Non-contact method of motion transformation | ♦ Requires print head area for the spring | ♦ IJ15 |
| Coiled actuator | A bend actuator is coiled to provide greater travel in a reduced chip area. | ♦ Increases travel<br>♦ Reduces chip area<br>♦ Planar implementations are relatively easy to fabricate. | ♦ Restricted to planar implementations due to extreme fabrication difficulty in other orientations. | ♦ IJ17, IJ21 |
| Flexure region near fixture | A bend actuator has a small region near the fixture point which flexes much more readily than the remainder of the actuator. The actuator flexing is effectively converted from an even coiling to an angular bend, resulting in greater travel of the actuator tip. | ♦ Simple means of increasing travel of a bend actuator | ♦ Care must be taken not to exceed the elastic limit in the flexure area<br>♦ Stress distribution is very uneven<br>♦ Difficult to accurately model with finite element analysis | ♦ IJ10, IJ19 |
| Catch | The actuator controls a small catch. The catch either enables or disables movement of an ink pusher which is controlled in a bulk manner. | ♦ Very low actuator energy<br>♦ Very small actuator size | ♦ Complex construction<br>♦ Requires external force<br>♦ Unsuitable for pigmented inks | ♦ IJ10 |

| | | | | |
|---|---|---|---|---|
| **Gears** | Gears can be used to increase travel at the expense of duration. Circular gears, rack and pinion, ratchets, and other gearing methods can be used. | ♦ Low force, low travel actuators can be used<br>♦ Can be fabricated using standard surface MEMS processes | ♦ Moving parts are required<br>♦ Several actuator cycles are required<br>♦ More complex drive electronics<br>♦ Complex construction<br>♦ Friction possible<br>♦ Stiction possible<br>♦ Wear possible<br>♦ Unsuitable for pigmented inks | ♦ IJ13 |
| **Buckle plate** | A buckle plate can be used to change a slow actuator into a fast motion. It can also convert a high force, low travel actuator into a high travel, medium force motion. | ♦ Very fast movement achievable | ♦ Must stay within elastic limits of the materials for long device life<br>♦ High stresses involved<br>♦ Generally high power requirement | ♦ S. Hirata et al, "An Ink-jet Head Using Diaphragm Microactuator", Proc. IEEE Micro Electro Mechanical Systems, Feb. 1996, pp 418-423.<br>♦ IJ18, IJ27 |
| **Tapered magnetic pole** | A tapered magnetic pole can increase travel at the expense of force. | ♦ Linearizes the magnetic force/distance curve | ♦ Complex construction | ♦ IJ14 |
| **Rotary impeller** | The actuator is connected to a rotary impeller. A small angular deflection of the actuator results in a rotation of the impeller vanes, which push the ink against stationary vanes and out of the nozzle. The ratio of force to travel of the actuator can be matched to the nozzle requirements by varying the number of impeller vanes. | ♦ High mechanical advantage | ♦ Complex construction<br>♦ Unsuitable for pigmented inks | ♦ IJ28 |

## Actuator motion

| Actuator motion | Description | Advantages | Disadvantages | Examples |
|---|---|---|---|---|
| **Volume expansion** | The volume of the actuator changes, pushing the ink in all directions. | ◆ The actuator is relatively non-directions, so placement of the nozzle relative to the actuator is not critical | ◆ High energy is typically required to achieve volume expansion | ◆ Hewlett-Packard Thermal Ink Jet<br>◆ Canon Bubblejet |
| **Linear, normal to chip surface** | The actuator moves in a direction normal to the print head surface. The nozzle is typically in the line of movement. | ◆ Efficient coupling to ink drops ejected normal to the surface | ◆ High fabrication complexity may be required to achieve perpendicular motion | ◆ IJ01, IJ02, IJ04, IJ07<br>◆ IJ11, IJ14 |
| **Linear, parallel to chip surface** | The actuator moves parallel to the print head surface. Drop ejection may still be normal to the surface. | ◆ Suitable for planar fabrication | ◆ Fabrication complexity<br>◆ Friction<br>◆ Stiction | ◆ IJ12, IJ13, IJ15 |
| **Membrane push** | An actuator with a high force but small area is used to push a stiff membrane which is in contact with the ink. | ◆ The effective area of the actuator becomes the membrane area | ◆ Fabrication complexity<br>◆ Actuator size<br>◆ Difficulty of integration in a VLSI process | ◆ 1982 Howkins USP 4,459,601 |
| **Rotary** | The actuator causes the rotation of some element, such a grill or impeller | ◆ Rotary levers may be used to increase travel<br>◆ Small chip area requirements | ◆ Device complexity<br>◆ May have friction at a pivot point | ◆ IJ05, IJ08, IJ13, IJ28 |
| **Bend** | The actuator bends when energized. This may be due to differential thermal expansion, piezoelectric expansion, magnetostriction, or other form of relative dimensional change. | ◆ A very small change in dimensions can be converted to a large motion. | ◆ Requires the actuator to be made from at least two distinct layers, or to have a thermal difference across the actuator | ◆ 1970 Kyser et al USP 3,946,398<br>◆ 1973 Stemme USP 3,747,120<br>◆ IJ03, IJ06, IJ09, IJ10<br>◆ IJ19, IJ23, IJ24, IJ25<br>◆ IJ29, IJ30 |
| **Straighten** | The actuator is normally bent, and straightens when energized. | ◆ Can be used with shape memory alloys where the austenic phase is planar | ◆ Requires careful balance of stresses to ensure that the quiescent bend is accurate | ◆ IJ26 |
| **Shear** | Energizing the actuator causes a shear motion in the actuator material. | ◆ Can increase the effective travel of piezoelectric actuators | ◆ Not readily applicable to other actuator mechanisms | ◆ 1985 Fishbeck USP 4,584,590 |
| **Radial constriction** | An ink reservoir is squeezed by the actuator, forcing the ink from a constricted nozzle. | ◆ Relatively easy to fabricate single nozzles from glass tubing as macroscopic structures | ◆ High force required<br>◆ Inefficient<br>◆ Difficult to integrate with VLSI processes | ◆ 1970 Zoltan USP 3,683,212 |
| **Coil / uncoil** | A coiled actuator uncoils or coils more tightly. The motion of the free end of the actuator ejects the ink. | ◆ Easy to fabricate as a planar VLSI process<br>◆ Small area required, therefore low cost | ◆ Difficult to fabricate for non-planar devices<br>◆ Poor out-of-plane stiffness | ◆ IJ17, IJ21 |
| **Bow** | The actuator bows (or buckles) in the middle when energized. | ◆ Can increase the speed of travel<br>◆ Mechanically rigid | ◆ Maximum travel is constrained<br>◆ High force required | ◆ IJ16, IJ18, IJ27 |
| **Push-Pull** | A shutter is controlled by two actuators. One actuator pulls the shutter, and the other pushes it. | ◆ The structure is pinned at both ends, so has a high out-of-plane rigidity | ◆ Not readily suitable for ink jets which directly push the ink | ◆ IJ18 |
| **Curl/ uncurl** | A set of actuators curl up to reduce the volume of ink that they enclose. | ◆ Good ink flow to the region behind the actuator increases efficiency | ◆ Design complexity | ◆ IJ20 |
| **Iris** | Multiple vanes enclose a volume of ink. These simultaneously rotate, reducing the volume between the vanes. | ◆ Good efficiency | ◆ High fabrication complexity<br>◆ Not suitable for pigmented inks | ◆ IJ22 |

## Nozzle refill method

| Nozzle refill method | Description | Advantages | Disadvantages | Examples |
|---|---|---|---|---|
| **Surface tension** | This is the normal way that ink jets are refilled. After the actuator is energized, it typically returns rapidly to its normal position. This rapid return sucks in air through the nozzle opening. The ink surface tension at the nozzle then exerts a small force restoring the meniscus to a minimum area. This force refills the nozzle. | ◆ Fabrication simplicity<br>◆ Operational simplicity | ◆ Low speed<br>◆ Surface tension force relatively small compared to actuator force<br>◆ Long refill time usually dominates the total ink jet | ◆ Thermal ink jet<br>◆ Piezo-electric ink jet<br>◆ IJ01, IJ02, IJ03, IJ04<br>◆ IJ05, IJ05, IJ07, IJ10<br>◆ IJ11, IJ12, IJ14, IJ16<br>◆ IJ20, IJ22, IJ23, IJ24<br>◆ IJ25, IJ26, IJ27, IJ28<br>◆ IJ29, IJ30 |
| **Shuttered oscillating ink pressure** | Ink to the nozzle chamber is provided at a pressure which oscillates at twice the drop ejection frequency. When a drop is to be ejected, the shutter is opened for 3 half cycles: drop ejection, actuator return, and refill. The shutter is then closed to prevent the nozzle chamber emptying during the next negative pressure cycle. | ◆ High speed<br>◆ Low actuator energy, as the actuator need only open or close the shutter, instead of ejecting the ink drop | ◆ Requires common ink pressure oscillator<br>◆ May not be suitable for pigmented inks | ◆ IJ08, IJ13, IJ15, IJ17<br>◆ IJ18, IJ19, IJ21 |
| **Refill actuator** | After the main actuator has ejected a drop a second (refill) actuator is energized. The refill actuator pushes ink into the nozzle chamber. The refill actuator returns slowly, to prevent its return from emptying the chamber again. | ◆ High speed, as the nozzle is actively refilled | ◆ Requires two independent actuators per nozzle | ◆ IJ09 |
| **Positive ink pressure** | The ink is held a slight positive pressure. After the ink drop is ejected, the nozzle chamber fills quickly as surface tension and ink pressure both operate to refill the nozzle. | ◆ High refill rate, therefore a high drop repetition rate is possible | ◆ Surface spill is more likely<br>◆ Highly hydrophobic print head surfaces are required | ◆ LIFT<br>◆ Alternative for:<br>◆ IJ01, IJ02, IJ03, IJ04<br>◆ IJ05, IJ05, IJ07, IJ10<br>◆ IJ11, IJ12, IJ14, IJ16<br>◆ IJ20, IJ22, IJ23, IJ24<br>◆ IJ25, IJ26, IJ27, IJ28<br>◆ IJ29, IJ30 |

## Nozzle plate construction

| Nozzle plate construction | Description | Advantages | Disadvantages | Examples |
|---|---|---|---|---|
| **Electroformed nickel** | A nozzle plate is separately fabricated from electroformed nickel, and bonded to the print head chip. | ◆ Fabrication simplicity | ◆ High temperatures and pressures are required to bond nozzle plate<br>◆ Minimum thickness constraints<br>◆ Differential thermal expansion | ◆ Hewlett Packard Thermal Ink Jet |
| **Laser ablated or drilled polymer** | Individual nozzle holes are ablated by an intense UV laser in a nozzle plate, which is typically a polymer such as polyimide or polysulphone | ◆ No masks required<br>◆ Can be quite fast<br>◆ Some control over nozzle profile is possible<br>◆ Equipment required is relatively low cost | ◆ Each hole must be individually formed<br>◆ Special equipment required<br>◆ Slow where there are many thousands of nozzles per print head<br>◆ May produce thin burrs at exit holes | ◆ Canon Bubblejet<br>◆ 1988 Sercel et al., SPIE, Vol. 998 Excimer Beam Applications, pp. 76-83<br>◆ 1993 Watanabe et al., USP 5,208,604 |
| **Silicon micro-machined** | A separate nozzle plate is micromachined from single crystal silicon, and bonded to the print head wafer. | ◆ High accuracy is attainable | ◆ Two part construction<br>◆ High cost<br>◆ Requires precision alignment<br>◆ Nozzles may be clogged by adhesive | ◆ K. Bean, IEEE Transactions on Electron Devices, Vol. ED-25, No. 10, 1978, pp 1185-1195<br>◆ Xerox 1990 Hawkins et al., USP 4,899,181 |
| **Glass capillaries** | Fine glass capillaries are drawn from glass tubing. This method has been used for making individual nozzles, but is difficult to use for bulk manufacturing of print heads with thousands of nozzles. | ◆ No expensive equipment required<br>◆ Simple to make single nozzles | ◆ Very small nozzle sizes are difficult to form<br>◆ Not suited for mass production | ◆ 1970 Zoltan USP 3,683,212 |
| **Monolithic, surface micro-machined using VLSI lithographic processes** | The nozzle plate is deposited as a layer using standard VLSI deposition techniques. Nozzles are etched in the nozzle plate using VLSI lithography and etching. | ◆ High accuracy (<1 μm)<br>◆ Monolithic<br>◆ Low cost<br>◆ Existing processes can be used | ◆ Requires sacrificial layer under the nozzle plate to form the nozzle chamber<br>◆ Surface is fragile to the touch | ◆ LIFT<br>◆ IJ01, IJ02, IJ04, IJ11<br>◆ IJ12, IJ17, IJ18, IJ20<br>◆ IJ22, IJ24, IJ27, IJ28<br>◆ IJ29, IJ30 |
| **Monolithic, etched through substrate** | The nozzle plate is a buried etch stop in the wafer. Nozzle chambers are etched in the front of the wafer, and the wafer is thinned from the back side to the etch stop. Nozzles are then etched in the etch stop layer. | ◆ High accuracy (<1 μm)<br>◆ Monolithic<br>◆ Low cost<br>◆ No differential expansion | ◆ Requires long etch times<br>◆ Requires a support wafer | ◆ IJ03, IJ05, IJ06, IJ07<br>◆ IJ08, IJ09, IJ10, IJ13<br>◆ IJ14, IJ15, IJ16, IJ19<br>◆ IJ21, IJ23, IJ25, IJ26 |
| **No nozzle plate** | Various methods have been tried to eliminate the nozzles entirely, to prevent nozzle clogging. These include thermal bubble mechanisms and acoustic lens mechanisms | ◆ No nozzles to become clogged | ◆ Difficult to control drop position accurately<br>◆ Crosstalk problems | ◆ Ricoh 1995 Sekiya et al USP 5,412,413<br>◆ 1993 Hadimioglu et al EUP 550,192<br>◆ 1993 Elrod et al EUP 572,220 |
| **Nozzle slit instead of individual nozzles** | The elimination of nozzle holes and replacement by a slit encompassing many actuator positions reduces nozzle clogging, but increases crosstalk due to ink surface waves | ◆ No nozzles to become clogged | ◆ Difficult to control drop position accurately<br>◆ Crosstalk problems | ◆ 1989 Saito et al USP 4,799,068 |

## Drop ejection direction

| Ejection direction | Description | Advantages | Disadvantages | Examples |
|---|---|---|---|---|
| **Edge** ('edge shooter') | Ink flow is along the surface of the chip, and ink drops are ejected from the chip edge. | ♦ Simple construction<br>♦ No silicon etching required<br>♦ Good heat sinking via substrate<br>♦ Mechanically strong<br>♦ Ease of chip handing | ♦ Nozzles limited to edge<br>♦ High resolution is difficult<br>♦ Fast color printing requires one print head per color | ♦ Canon Bubblejet 1979 Endo et al GB patent 2,007,162<br>♦ Xerox heater-in-pit 1990 Hawkins et al USP 4,899,181 |
| **Surface** ('roof shooter') | Ink flow is along the surface of the chip, and ink drops are ejected from the chip surface, normal to the plane of the chip. | ♦ No bulk silicon etching required<br>♦ Silicon can make an effective heat sink<br>♦ Mechanical strength | ♦ Maximum ink flow is severely restricted | ♦ Hewlett-Packard TIJ 1982 Vaught et al USP 4,490,728<br>♦ IJ02, IJ11, IJ12, IJ20<br>♦ IJ22 |
| **Through chip, forward** ('up shooter') | Ink flow is through the chip, and ink drops are ejected from the front surface of the chip. | ♦ High ink flow<br>♦ Suitable for pagewidth print heads<br>♦ High nozzle packing density therefore low manufacturing cost | ♦ Requires bulk silicon etching | ♦ LIFT<br>♦ IJ04, IJ17, IJ18, IJ24<br>♦ IJ27, IJ28, IJ29, IJ30 |
| **Through chip, reverse** ('down shooter') | Ink flow is through the chip, and ink drops are ejected from the rear surface of the chip. | ♦ High ink flow<br>♦ Suitable for pagewidth print heads<br>♦ High nozzle packing density therefore low manufacturing cost | ♦ Requires wafer thinning<br>♦ Requires special handling during manufacture | ♦ IJ01, IJ03, IJ05, IJ06<br>♦ IJ07, IJ08, IJ09, IJ10<br>♦ IJ13, IJ14, IJ15, IJ16<br>♦ IJ19, IJ21, IJ23, IJ25<br>♦ IJ26 |
| **Through actuator** | Ink flow is through the actuator, which is not fabricated as part of the same substrate as the drive transistors. | ♦ Suitable for piezo-electric print heads | ♦ Pagewidth print head requires thousands of connections to drive electronics<br>♦ Cannot be manufactured in standard CMOS fabs<br>♦ Complex assembly required | ♦ Epson Stylus<br>♦ Tektronix |

Ink type

| Ink type | Description | Advantages | Disadvantages | Examples |
|---|---|---|---|---|
| Aqueous, dye | Water based ink which typically contains: water, dye, surfactant, humectant, and biocide.<br>Modern ink dyes have high water-fastness, light fastness | ♦ Environmentally friendly<br>♦ No odor | ♦ Slow drying<br>♦ Corrosive<br>♦ Bleeds on paper<br>♦ May strikethrough<br>♦ Cockles paper | ♦ Most existing ink jets<br>♦ IJ01, IJ02, IJ03, IJ04<br>♦ IJ05, IJ06, IJ07, IJ08<br>♦ IJ09, IJ10, IJ11, IJ12<br>♦ IJ13, IJ14, IJ15, IJ16<br>♦ IJ17, IJ18, IJ19, IJ20<br>♦ IJ21, IJ22, IJ23, IJ24<br>♦ IJ25, IJ26, IJ27, IJ28<br>♦ IJ29, IJ30 |
| Aqueous, pigment | Water based ink which typically contains: water, pigment, surfactant, humectant, and biocide.<br>Pigments have an advantage in reduced bleed, wicking and strikethrough. | ♦ Environmentally friendly<br>♦ No odor<br>♦ Reduced bleed<br>♦ Reduced wicking<br>♦ Reduced strikethrough | ♦ Slow drying<br>♦ Corrosive<br>♦ Pigment may clog nozzles<br>♦ Pigment may clog actuator mechanisms<br>♦ Cockles paper | ♦ IJ02, IJ04, IJ21, IJ26<br>♦ IJ27, IJ30 |
| Methyl Ethyl Ketone (MEK) | MEK is a highly volatile solvent used for industrial printing on difficult surfaces such as aluminum cans | ♦ Very fast drying<br>♦ Prints on various substrates such as metals and plastics | ♦ Odorous<br>♦ Flammable | ♦ IJ01, IJ02, IJ03, IJ04<br>♦ IJ05, IJ06, IJ07, IJ08<br>♦ IJ09, IJ10, IJ11, IJ12<br>♦ IJ13, IJ14, IJ15, IJ16<br>♦ IJ17, IJ18, IJ19, IJ20<br>♦ IJ21, IJ22, IJ23, IJ24<br>♦ IJ25, IJ26, IJ27, IJ28<br>♦ IJ29, IJ30 |
| Phase change (hot melt) | The ink is solid at room temperature, and is melted in the print head before jetting. Hot melt inks are usually wax based, with a melting point around 80 °C. After jetting the ink freezes almost instantly upon contacting the print medium or a transfer roller. | ♦ No drying time- ink instantly freezes on the print medium<br>♦ Almost any print medium can be used<br>♦ No paper cockle occurs<br>♦ No wicking occurs<br>♦ No bleed occurs<br>♦ No strikethrough occurs | ♦ High viscosity<br>♦ Printed ink typically has a 'waxy' feel<br>♦ Printed pages may 'block'<br>♦ Ink temperature may be above the curie point of permanent magnets<br>♦ Ink heaters consume power<br>♦ Long warm-up time | ♦ Tektronix<br>♦ 1989 Nowak USP 4,820,346<br>♦ IJ01, IJ02, IJ03, IJ04<br>♦ IJ05, IJ06, IJ08, IJ09<br>♦ IJ11, IJ12, IJ13, IJ14<br>♦ IJ15, IJ16, IJ17, IJ18<br>♦ IJ19, IJ20, IJ21, IJ22<br>♦ IJ23, IJ24, IJ25, IJ26<br>♦ IJ27, IJ28, IJ29, IJ30 |
| Oil | Oil based inks are extensively used in offset printing. They have advantages in improved characteristics on paper (especially no wicking or cockle). Oil soluble dies and pigments are required. | ♦ High solubility medium for some dyes<br>♦ Does not cockle paper<br>♦ Does not wick through paper | ♦ High viscosity: this is a significant limitation for use in ink jets, which usually require a low viscosity. Some short chain and multi-branched oils have a sufficiently low viscosity.<br>♦ Slow drying | ♦ IJ01, IJ02, IJ03, IJ04<br>♦ IJ05, IJ06, IJ07, IJ08<br>♦ IJ09, IJ10, IJ11, IJ12<br>♦ IJ13, IJ14, IJ15, IJ16<br>♦ IJ17, IJ18, IJ19, IJ20<br>♦ IJ21, IJ22, IJ23, IJ24<br>♦ IJ25, IJ26, IJ27, IJ28<br>♦ IJ29, IJ30 |
| Microemulsion | A microemulsion is a stable, self forming emulsion of oil, water, and surfactant. The characteristic drop size is less than 100 nm, and is determined by the preferred curvature of the surfactant. | ♦ Stops ink bleed<br>♦ High dye solubility<br>♦ Water, oil, and amphiphilic soluble dies can be used<br>♦ Can stabilize pigment suspensions | ♦ Viscosity higher than water<br>♦ Cost is slightly higher than water based ink<br>♦ High surfactant concentration required (around 5%) | ♦ IJ01, IJ02, IJ03, IJ04<br>♦ IJ05, IJ06, IJ07, IJ08<br>♦ IJ09, IJ10, IJ11, IJ12<br>♦ IJ13, IJ14, IJ15, IJ16<br>♦ IJ17, IJ18, IJ19, IJ20<br>♦ IJ21, IJ22, IJ23, IJ24<br>♦ IJ25, IJ26, IJ27, IJ28<br>♦ IJ29, IJ30 |

## APPENDIX C – IJ SERIES INK JETS

This appendix sets out a series of novel arrangements of ink jets constructed in accordance with the physical variables as set out in appendix B. Each of these ink jets include features that can be combined with other inkjets in accordance with requirements.

### IJ01 DIRECT PLUNGER ELECTROMAGNETIC INK JET

Ink Jet IJ01 is described with reference to the accompanying drawings in which:

Fig. C01.1 illustrates a cross section of a single nozzle and actuator of a 1200 dpi print head, shown firing an ink drop.
Fig. C01.2 illustrates an exploded view of the nozzle.

IJ01 is a direct firing electromagnetic ink jet, where an electromagnet attracts a soft magnetic plunger which ejects the ink. When data signals distributed on the print head indicate that a particular nozzle is to eject a drop of ink, the drive transistor for that nozzle is turned on. This energizes a solenoid, which induces a magnetic field in a nickel iron fixed plate and movable plunger. When the solenoid is energized, the plunger is attracted to the fixed plate across a gap. The plunger pushes against the ink, creating a high pressure in the nozzle chamber, causing ink to be squirted out of the nozzle. Ink trapped between the plunger and the solenoid is squirted out of the holes in the top of the plunger. This prevents trapped ink increasing the pressure on the plunger, and thereby requiring more magnetic force to move the plunger. After approximately 2 μs, the current to the solenoid is turned off. At the same time, or at a slightly later time, a reverse current is applied, approximately half of the forward current. As the plunger will carry some residual magnetism, this causes the plunger to move backwards towards its nominal position. A spring also helps to return the plunger. The reverse current is turned off before the magnetization of the plunger is reversed, which would cause the plunger be attracted to the fixed plate again. The return of the plunger to its quiescent position causes a low pressure in the ink chamber. This causes ink to begin flowing from the ejected drop back into the nozzle, and also ingests air into the chamber. The forward velocity of the drop and backward velocity of the ink in the chamber are resolved by the ink drop breaking off from the ink in the nozzle. The ink drop then continues to travel towards the recording medium under its own momentum. The nozzle refills due to the surface tension of the ink at the nozzle tip. Shortly after the time of drop breakoff, the meniscus at the nozzle tip is approximately a concave hemisphere. The surface tension exerts a net forward force on the ink, which results in the nozzle refilling. The repetition rate of the ink jet is principally determined by the nozzle refill time, which will be approximately 100 μs, depending upon device geometry, ink surface tension, and the volume of the ejected drop.

In an alternative arrangement ink in the region of the solenoid is also squirted out of the nozzle, through a series of posts that complete the magnetic circuit. The ink around the solenoid has a higher fluidic resistance before reaching the nozzle than ink directly between the nozzle and the plunger. It will therefore exert a greater pressure on the plunger than if the ink were allowed to escape, requiring greater magnetic force to move the plunger. However, this ink adds to the quantity of ink which is squirted out of the nozzle, thereby allowing a smaller plunger travel, and a smaller gap between the plunger and the fixed magnetic plate. This smaller gap means that substantially more magnetic force is available to move the plunger.

The drop firing rate is around 7 KHz. The ink jet head is suitable for fabrication as a monolithic pagewide print head. The print head has a 'down shooter' configuration.

## IJ02 ELECTROSTATIC INK JET

Ink Jet IJ02 is described with reference to the accompanying drawings in which:

Fig. C02.1 illustrates a single nozzle and actuator of a 1600 dpi print head. The central hole is the nozzle, with a 8 μm radius. The smaller holes are to allow etching of the sacrificial layers during fabrication. Ink flows accross the chip surface, and enters the nozzle chamber via ports at the periphery.
Fig. C02.2 illustrates a cross section detail showing the nozzle wall, concertina ring, electrostatic plates, and air vents.

IJ02 is a reverse firing electrostatic ink jet. To prepare for firing a drop, a voltage difference is applied to a pair of parallel conductive plates below the nozzle chamber. These conductive plates are large relative to the nozzle, and are separated by a small air gap. The applied voltage induces charges on the plates proportional to the capacitance of the two plates. The opposite charges result in an attractive electrostatic force, which causes the upper plate to move towards the lower plate. The upper plate is suspended on a concertina spring, so movement is achieved by bending the spring, not stretching the upper plate. The lower plate is embedded in the substrate material. The air between the two plates is vented to the outside, via air vents along the periphery of the nozzle chambers. The gap between the two plates is lined with a hydrophobic material, such as polytetrafluoroethylene (PTFE), which prevents any spilled ink from entering the air gap and filling it by capillarity. The PTFE lining also prevents stiction after sacrificial etching.

The voltage applied to the plates is maintained at least long enough for the nozzle chamber to refill, and replace the ink displaced by the movement of the upper plate. The nozzle refill is achieved by the surface tension of the ink at the nozzle, and requires approximately 100 μs, depending upon device geometry, ink surface tension, and the volume of the ejected drop.

To fire the drop, the voltage difference between the plates is removed. The elastic forces in the concertina spring cause the upper plate to move back to its normal position, displacing ink as it moves. The displaced ink is ejected from the nozzle.

The voltage is reapplied to the plates after about 5 μs, and maintained until the next drop is to be fired. The re-application of the voltage causes the upper plate to move down again while the ink drop is being ejected. This helps with the drop break-off process. The exact timing of the re-application of the voltage can be altered to minimize satellite drop formation.

The forward velocity of the drop and backward velocity of the upper electrostatic plate are resolved by the ink drop breaking off from the ink in the nozzle. The ink drop then continues to travel towards the recording medium under its own momentum. The repetition rate of the ink jet is principally determined by the nozzle refill time.

The drop firing rate is around 7 KHz. While relatively large, the ink jet head can be fabricated as a monolithic pagewide print head. The print head has a 'roof shooter' configuration.

## IJ03 PLANAR THERMOELASTIC BEND ACTUATOR INK JET

Ink Jet IJ03 is described with reference to the accompanying drawings in which:

Fig. C03.1 illustrates a view of a single 1600 dpi nozzle actuator from the inside of the ink reservoir. The top layer is ITO, the layer below that is aluminum, followed silicon nitride.
Fig. C03.2 illustrates a cross section of the nozzle and actuator. The bottom layer is boron doped silicon etch stop. The crystallographically etched nozzle chamber is shown in the silicon layer. The nozzle radius is 8μ m for 1600 dpi operation.

IJ03 has one thermoelastic bend actuator for each nozzle. The bend actuator is a planar layer of two materials separated by a small gap. It is mechanically fixed to the substrate at one end, and the other end is free to move.

The two materials comprising the actuator are electrically joined at the end which is mechanically free. The mechanically fixed end also provides the electrical connections between the drive circuitry and the two plates.

When energized, the actuator bends into the substrate towards a nozzle, pushing on ink in the nozzle chamber. This ink is pushed out of the nozzle, forming the ink drop. As the actuator is cooled by the ink, it returns to its nominal position, withdrawing ink from the nozzle and aiding in drop separation.

The actuator is composed of 4 layers:

1) A upper layer of a high resistivity material such as indium-tin oxide (ITO). This layer dissipates the vast majority of the electrical energy passed through the actuator as heat. The resultant temperature rise causes this layer to expand. The ideal characteristics of this layer include a high resistivity, a high Young's modulus, and a high coefficient of thermal expansion.
2) A gap. This gap maintains electrical and mechanical separation between the two conductive layers.
3) A lower layer of a high conductivity material, such as aluminium. As the resistance of this layer is very small, a negligible amount of heat is dissipated. The layer's characteristics should include a low resistivity, a low Young's modulus, and a low coefficient of thermal expansion.
4) A stiffening plate of silicon nitride approximately 80% of the length of the actuator is attached at the free end. This stiffener prevents the lower layer from bending evenly, forcing most of the bend to occur near the fixed end of the actuator. In this manner, a greater deflection of the actuator tip is achieved.

The print head is fabricated as a CMOS device fabricated on a wafer with an epitaxial layer of high concentration boron doped silicon, and a subsequent 10 μm thick layer of lightly doped epitaxial silicon. After CMOS processing (in the lightly doped epitaxial silicon) is complete, MEMS post processing is performed. This includes a nitride passivation layer, an aluminum layer, a sacrificial oxide layer, and an ITO layer. The sacrificial oxide is etched, followed by a crystallographic wet etch of silicon which forms the nozzle chamber as a pit in the front surface silicon. The wafer is then bonded to an ink channel wafer, and the entire back side of the wafer is etched until the boron doped layer is reached. The nozzle holes are then masked and plasma etched.

Not shown are the layers of passivation material required to prevent corrosion of the aluminum and ITO layers by the ink.

The drop firing rate is around 7 KHz. The ink jet head is suitable for fabrication as a monolithic pagewide print head. The print head has an 'roof shooter' configuration.

## IJ04 STACKED ELECTROSTATIC/PIEZOELECTRIC/ELECTROSTRICTIVE INK JET

Ink Jet IJ04 is described with reference to the accompanying drawings in which:

Fig. C04.1 illustrates a view of a single 1600 dpi nozzle and actuator stack. The actuator stack consists of 40 layers of elastomer sandwiched between electrodes.
Fig. C04.2 illustrates a cross section detail showing one end of the actuator stack, next to the silicon nitride chamber filter.

IJ04 uses a stack of electrostatic plates separated by an elastomer dielectric. Alternate plates are connected to one electrode, and the other plates are connected to another electrode. When a voltage is applied across the electrodes, alternate charges on alternate plates cause the plates to attract each other, compressing the elastomer. For efficient operation, manufacturability, and reliability of the actuator, the properties of the elastomer are critical. To achieve a small actuator size, the elastomer should have a very low bulk modulus, a high dielectric constant, and a high dielectric strength. For high manufacturability, it must be able to be applied in thin (approx. 0.2 μm) pin-hole free layers by spin coating, evaporation, or other means. It must also survive subsequent processing steps which may occur at elevated temperatures. For reliability, the elastomer should have high stability, and be chemically inert.

Materials such as styrene ethylene butylene styrene block copolymer (trade name C-Flex R70-190 from Consolidated) have a very low 100% modulus of 0.14 MPa, which is not much higher than air at

atmospheric pressure. However, this material is not ideal in all respects, as the dielectric constant and dielectric strength are in the normal polymeric range.

The voltage applied to the alternate plates is maintained at least long enough for the nozzle chamber to refill, and replace the ink displaced by the compression of the stack. The nozzle refill is achieved by the surface tension of the ink at the nozzle, and requires approximately 100 μs, depending upon device geometry, ink surface tension, and the volume of the ejected drop. To fire the drop, the voltage difference between the plates is removed. The elastic forces in the elastomer cause the stack to return to its normal volume, ejecting ink from the nozzle.

The voltage is reapplied to the plates after about 5 μs, and maintained until the next drop is to be fired. The re-application of the voltage causes the stack to compress again while the ink drop is being ejected. This helps with the drop break-off process. The exact timing of the re-application of the voltage can be altered to minimize satellite drop formation.

In an alternative version, thin films which respond mechanically to the electrostatic field of the electrodes can be used in place of the elastomer. Such materials include:

1) Piezoelectric materials such as PZT
2) Electrostrictive materials such as PLZT
3) Electrically switched transitions between ferroelectric (FE) to antiferroelectric (AFE) phases of a material such as PLZSnT.

The area of the stack is determined by the strain that the material achieves under the applied voltage. Piezoelectric and electrostrictive materials require a larger stack area due to the small strains achieved. With these materials, the stack can be made to expand as well as contract. With an expanding stack, the voltage is applied when the drop is to be fired. A stack which both expands and contracts (such as piezoelectric materials) can be driven with a bipolar voltage to achieve twice the peak-to-peak strain.

## IJ05 REVERSE SPRING LEVER ELECTROMAGNETIC INK JET

Ink Jet IJ05 is described with reference to the accompanying drawings in which:

Fig. C05.1 illustrates a single nozzle, plunger, spring and actuator of a 1600 dpi print head, viewed from inside the ink reservoir.
Fig. C05.2 illustrates an exploded view of a nozzle, plunger, spring and actuator.

IJ05 is an ink jet print head which has one magnetic actuator for each nozzle. Instead of directly ejecting a drop when the actuator solenoid is energized, the drop is ejected when the solenoid is turned off. The magnetic actuator is used to load a spring connected to a plunger. It is the return of the spring to its non-stressed position which moves the plunger and ejects the drop.

The advantages of this are that the plunger velocity is much more constant over the duration of the drop ejection stroke, and that the piston or plunger can be entirely removed from the ink chamber during the ink fill stage, reducing nozzle refill time.

When data signals distributed on the print head indicate that a particular nozzle is to eject a drop of ink, the drive transistor for that nozzle is turned on. This energizes a solenoid, which induces a magnetic field in fixed plate and a movable plunger, constructed from a soft magnetic material such as nickel iron (NiFe). The solenoid power is turned to the maximum current for long enough to move the plunger to its stop position (approximately 2μs). The piston is withdrawn from the nozzle chamber, drawing air into the chamber through the nozzle.

The solenoid current is turned to a 'keeper' level while the nozzle fills. The keeper power level is sufficient to maintain the movable pole against the stop. This will typically be substantially less than the maximum current level as the magnetic gap is at a minimum. During the 'keeper' phase, the meniscus at the nozzle tip is approximately a concave hemisphere. The surface tension exerts a net force on the ink, which results in the nozzle refilling, replacing the volume of the piston withdrawal with ink. This process takes approximately 100 μs.

The solenoid current is then reversed, at around half of the maximum current. The reversal is to demagnetize the magnetic plate and plunger, and to initiate the return of the piston to its nominal

position. The piston is accelerated to its nominal position both by the magnetic repulsion, and by the stressed spring. The force on the piston is greatest at the beginning of the stroke, and slows as the spring elastic stress falls to zero. As a result, the acceleration is high at the beginning of the stroke, and the ink velocity is much more uniform during the stroke. This results in an increased operating tolerance before ink flow over the print head front surface can occur. When the residual magnetism of the plunger is at a minimum, the solenoid reverse current is turned off. The piston continues to move towards the quiescent position. The piston overshoots due to its inertia. Overshoot in the piston movement achieves two things: greater ejected drop volume and velocity, and improved drop breakoff as the piston returns from overshoot to its quiescent position.

The piston return is caused by the spring, which is now stressed in the opposite direction. This motion 'sucks' some of the ink back into the nozzle, causing the ink ligament connecting the ink drop to the ink in the nozzle to thin. The forward velocity of the drop and backward velocity of the ink in the chamber are resolved by the ink drop breaking off from the ink in the nozzle. The piston is then at the quiescent position until the next drop ejection cycle.

The drop firing rate is around 7 KHz. The print head has a 'down shooter' configuration.

## IJ06 LORENZ PADDLE ELECTROMAGNETIC INK JET

Ink Jet IJ06 is described with reference to the accompanying drawings in which:

Fig. C06.1 illustrates a view of a single nozzle and actuator of a 1600 dpi print head seen from the inside of the ink reservoir. The paddle consists of a two layer copper coil which is embedded in silicon nitride. The print head is placed in a strong (1 Tesla) external magnetic field which is parallel to the plane of the wafer, and in the left-right plane of the paper as shown here.

Fig. C06.2 illustrates a cross section of the ink jet nozzle and actuator, showing the paddle deflected by the Lorenz force acting on the energized solenoid, which is in an external magnetic field.

IJ06 is a direct firing electromagnetic ink jet. Each nozzle has an associated ink chamber, which is etched into the substrate. On the opposite side of the chamber is a paddle which contains a solenoid. The entire print head is in a constant magnetic field, which may be provided by a permanent magnet, configuration of permanent magnets, or an electromagnet. For optimum efficiency, the constant magnetic field should be as strong as practical. Magnetic fields slightly greater than 1 Tesla can be achieved with neodymium iron boron (NdFeB) permanent magnets.

When data signals distributed on the print head indicate that a particular nozzle is to eject a drop of ink, the drive transistor for that nozzle is turned on. This energizes the solenoid. The solenoid current interacts with the fixed magnetic field, producing a torque on the paddle. One side of the paddle moves towards the nozzle, pushing ink out of the nozzle. The other side of the paddle has a force directed away from the nozzle. The paddle is asymmetric, with the side that moves towards the nozzle being longer in relation to the effective fulcrum, thereby providing more torque.

When the solenoid is energized, the paddle pushes against the ink, creating a high pressure in the nozzle chamber. This high pressure causes ink to be squirted out of the nozzle. After approximately 2 μs, the current to the solenoid is turned off. The springs connecting the paddle to the substrate quickly return of the paddle to its quiescent position. This causes ink to begin flowing from the ejected drop back into the nozzle, and also ingests air into the chamber. The forward velocity of the drop and backward velocity of the ink in the chamber are resolved by the ink drop breaking off from the ink in the nozzle. The ink drop then continues to travel towards the recording medium under its own momentum. The nozzle refills due to the surface tension of the ink at the nozzle tip. The repetition rate of the ink jet is principally determined by the nozzle refill time, which will be approximately 100 μs, depending upon device geometry, ink surface tension, and the volume of the ejected drop.

Two springs provide mechanical connection of the paddle to the substrate. A copper wire embedded in each spring electrically connects the solenoid to the drive circuitry.

The drop firing rate is around 7 KHz. The ink jet head is suitable for fabrication as a monolithic pagewide print head. The print head has a 'down shooter' configuration.

## IJ07 PERMANENT MAGNET ELECTROMAGNETIC INK JET

Ink Jet IJ07 is described with reference to the accompanying drawings in which:

Fig. C07.1 illustrates a cross section of a single nozzle and actuator of a 1600 dpi print head after the permanent magnet plunger has returned to its quiescent position after firing a drop.
Fig. C07.2 illustrates an exploded view of a single nozzle and actuator.

IJ07 is a direct firing ink jet print head. For each nozzle there is a solenoid and a permanent magnet piston. For maximum efficiency, a rare earth magnet such as neodymium iron boron (NdFeB) is used. As thousands of magnetic pistons are required in a pagewidth print head, they are fabricated using VLSI thin film deposition and etching techniques. The pistons are magnetized by placing the completed wafer in an intense magnetic field. When the solenoid is energized, it attracts the magnetic piston, causing ink to be ejected from the nozzle. After approximately 2 μs, the current to the solenoid is turned off. A spring attached to the piston returns it to its nominal quiescent position. This causes the ink drop to break off from the ink in the nozzle, and move towards the print medium. The nozzle refills due to the surface tension of the ink at the nozzle tip. In the quiescent position the piston is fully removed from the nozzle chamber, allowing faster refill.

The print head is fabricated as a CMOS device fabricated on a wafer with an epitaxial layer of heavily boron doped silicon, and a subsequent 10 μm thick layer of lightly doped epitaxial silicon. After CMOS processing (in the lightly doped epitaxial silicon) is complete, MEMS post processing is performed. This includes a nitride passivation layer, a copper coil layer, a sacrificial oxide layer, a second nitride passivation layer, a NdFeB layer, and a third nitride layer which provides both passivation and the spring. The sacrificial oxide is etched, followed by a crystallographic wet etch of silicon which forms the nozzle chamber as a pit in the front surface silicon. The wafer is then bonded to an ink channel wafer, and the entire back side of the wafer is etched until the boron doped layer is reached. The nozzle holes are then masked and plasma etched in the boron doped layer.

The drop firing rate is around 7 KHz. The ink jet head is suitable for fabrication as a monolithic pagewide print head. The print head has a 'down shooter' configuration.

## IJ08 PLANAR SWING GRILL ELECTROMAGNETIC INK JET

Ink Jet IJ08 is described with reference to the accompanying drawings in which:

Fig. C08.1 illustrates a single nozzle and actuator of a 1600 dpi ink jet viewed from the inside of the ink reservoir. The nozzle actuator is in the quiescent position, with the shutter grill blocking ink access to the nozzle chamber.
Fig. C08.2 illustrates an exploded view of the nozzle.

IJ08 is a magnetically actuated ink jet which uses a pivoting lever to increase the travel of a shutter grill.

An oscillating ink pressure is used to eject ink from the nozzles. Each nozzle has an associated shutter grill, which normally blocks the slots in a fixed grill over the nozzle chamber. The shutter grill is moved to avoid blocking the fixed grill slots by the electromagnetic actuator (a solenoid) whenever an ink drop is to be fired.

The solenoid attracts a soft magnetic (NiFe) bar. The bar is attracted by the solenoid on both sides of a central point, about which it rotates approximately 4 degrees. The bar is attached to the shutter grill, which rotates with the bar, exposing slots in the fixed grill. A one micron movement of the bar ends towards the solenoid results in an approximately 6 micron movement at the outside rim of the shutter grill. This dramatically improves the efficiency of the system, as the magnetic field falls off strongly with distance.

The surface of the wafer is directly immersed in the ink reservoir, or in relatively large ink channels. An ultrasonic transducer (for example, a piezoelectric transducer) is positioned in the reservoir. The transducer oscillates the ink pressure at approximately 100 KHz. The ink pressure oscillation is sufficient that ink drops would be ejected from the nozzle were it not blocked by the shutter.

When data signals distributed on the print head indicate that a particular nozzle is to eject a drop of ink, the drive transistor for that nozzle is turned on. This energizes the actuator, which moves the shutter so that it is not blocking the ink chamber. The peak of the ink pressure variation causes the ink to be squirted out of the nozzle. As the ink pressure goes negative, ink is drawn back into the nozzle, causing drop break-off. The shutter is kept open until the nozzle is refilled on the next positive pressure cycle. It is then shut to prevent the ink from being withdrawn from the nozzle on the next negative pressure cycle.

Each drop ejection takes two ink pressure cycles, resulting in a 50 KHz drop firing rate.

The amplitude of the ultrasonic transducer can be altered in response to the viscosity of the ink (which is typically affected by temperature), and the number of drops which are to be ejected in the current cycle. This amplitude adjustment can be used to maintain consistent drop size in varying environmental conditions.

The nozzle chamber is formed using an anisotropic crystallographic etch of the silicon substrate. Etchant access to the substrate is via the slots in the fixed grill. The device is manufactured on <100> silicon with a buried boron etch stop layer.

## IJ09 PUMP ACTION REFILL INK JET

Ink Jet IJ09 is described with reference to the accompanying drawings in which:

Fig. C09.1 illustrates a single nozzle and actuator of a 1600 dpi print head, showing the two thermal bend actuators. The view is from the inside of the ink reservoir.
Fig. C09.2 illustrates a cross section of quiescent position.
Fig. C09.3 illustrates a drop firing actuator (A ) energized.
Fig. C09.4 illustrates a drop break-off and bubble ingestion.
Fig. C09.5 illustrates a refill actuator (A) energized.
Fig. C09.6 illustrates a chamber full during slow return of A2.
Fig. C09.7 illustrates the combined activation of A1 and A2.

IJ09 has two thermoelastic bend actuator for each nozzle. One actuator ($A_1$) fires the drop, and the other actuator ($A_2$) refills the nozzle. With this method, a high drop repetition rate of approximately 50 KHz can be obtained. This is because there is no need to wait around 100 μs for surface tension to refill the nozzle after each drop is fired.

When a drop is to be fired, $A_1$ is energized by passing a current through an electrothermal heater embedded near the top surface of the actuator. The actuator is made of polytetrafluoroethylene (PTFE). PTFE has a very high coefficient of thermal expansion (approximately 770 x $10^{-6}$, or around 380 times that of silicon). The top region of the actuator is heated, but the bottom region is not. Differential thermal expansion causes the actuator to bend downwards, pushing on the ink in the ink chamber and expelling a drop from the nozzle.

When the heater current is turned off, the paddle begins to return to its quiescent position. The paddle return 'sucks' some of the ink back into the nozzle, causing the ink drop to break off from the ink in the nozzle. At the same time, air is ingested into the nozzle chamber through the nozzle. At this time, $A_2$ is actuated , causing the nozzle to refill. The power to $A_2$ is released slowly, so that the return of $A_2$ to its quiescent position matches the rate of refill of the ink chamber by surface tension. In this manner, the nozzle chamber stays approximately full during the relatively long refill period. High printing speeds are achieved by allowing the next drop to be fired at any time during the slow return of $A_2$. This is done by firing $A_1$ again. A new drop can be fired shortly after the previous drop, as the nozzle chamber remains full after $A_2$ is fired. $A_2$ is released approximately 1 μs after $A_1$ is fired the second time. The nozzle is refilled a second time by firing $A_2$ again after $A_1$ returns to its quiescent position. The exact timing, energy, and duration of the pulses to $A_1$ and $A_2$ are controlled to compensate for the minor drop size differences resulting from firing A1 at different times during the return of $A_2$.

The ink jet head is suitable for fabrication as a monolithic pagewide print head. The print head has an 'down shooter' configuration. The PTFE surfaces are rendered hydrophilic by high energy plasma bombardment in an ammonia atmosphere. Alternatively, a hydrophilic polymer can be used, albeit with somewhat lower efficiency.

## IJ10 PULSED MAGNETIC FIELD INK JET

Ink Jet IJ10 is described with reference to the accompanying drawings in which:

Fig. C10.1 illustrates a cross section of a single nozzle and actuator of a 1600 dpi print head, showing the passive magnetic paddle, and the thermal bend actuator catch. The catch is actuated, preventing the paddle from moving in response to the pulsed magnetic field. The silicon nitride spring is obscured behind the paddle.

Fig. C10.2 illustrates a cross section showing the catch turned off, allowing the paddle to be moved by the pulsed magnetic filed, thereby expelling a drop.

IJ10 utilizes an external pulsed magnetic field to attract a spring loaded paddle which pushes ink out of the nozzle. The pulsed magnetic field acts upon all of the paddles (one per nozzle) in the print head simultaneously. When unconstrained, each paddle fires a drop of ink for each pulse of the magnetic field. The magnetic field is pulsed when any drop is to be fired.

For those nozzles which are not to fire a drop, the catch is actuated, preventing the paddle from moving in response to the magnetic pulse.

The catch is actuated by a small electrothermal bend actuator. The forces required are only a few μN, and the displacement required is only a few μm. Therefore, the energy required to actuate the catch is very small. The energy required to eject the ink drops is provided by the external pulsed electromagnet. The magnetic material in the paddle can be either a soft ferromagnetic material such as nickel iron (NiFe), or a permanent magnetic material such as neodymium iron boron (NdFeB). If a permanent magnetic material is used, the material should be magnetized during wafer processing, after the last high temperature processing step.

Unlike most other ink jet types, the actuator is turned on when a drop is not to be fired, rather than when it is to be fired. This means that the number of drops fired is not proportional to the energy dissipated in the actuators. As a result, self-cooling by the ejected ink drops cannot be relied upon. Fortunately, the power consumption of the thermal actuators is so low that power dissipation is not a significant problem with this type of print head.

After a drop is fired, the nozzle refills due to the surface tension of the ink at the nozzle tip. The repetition rate of the ink jet is principally determined by the nozzle refill time, which will be approximately 100 μs, depending upon device geometry, ink surface tension, and the volume of the ejected drop.

The drop firing rate is around 7 KHz. The ink jet head is suitable for fabrication as a monolithic pagewide print head. The print head has a 'down shooter' configuration.

## IJ11 TWO COIL REVERSE FIRING ELECTROMAGNETIC INK JET

Ink Jet IJ11 is described with reference to the accompanying drawings in which:

Fig. C11.1 illustrates a single nozzle, static coil, and moving coil of a 1600 dpi print head.

Fig. C11.2 illustrates an exploded view of a nozzle, moving coil, and static coil. The nozzle chamber is fabricated from silicon nitride using an oxide sacrificial layer. Ink flow into the chamber is via slots in the rim of the chamber.

IJ11 is an ink jet print head which has a static coil and a movable coil for each nozzle. When energized, the static and movable coils are attracted towards each other, 'loading' a spring. The drop is ejected from the nozzle when the coils are de-energized. The movable coil forms a plunger, and is rapidly returned to its quiescent position by the loaded spring.

The use of coils without soft magnetic cores simplifies fabrication of the device, as no unusual materials such as NiFe are required. Also, there is no magnetic material which needs to be demagnetized. The disadvantage of a coil arrangement without soft magnetic materials is low magnetic flux density, requiring higher currents to achieve equivalent forces.

The coil metal is preferably copper, for high conductivity and increased resistance to electromigration. The copper coils are encased in silicon nitride for corrosion resistance and mechanical support.

When data signals distributed on the print head indicate that a particular nozzle is to eject a drop of ink, the drive transistor for that nozzle is turned on. This energizes both the static coil and the moving coil. The coils are energized for long enough for the moving coil to reach its stop position (approximately 2μs).

The coil current is turned to a 'keeper' level while the nozzle refills. The keeper power level is substantially less than the maximum current level because the magnetic gap is at a minimum when the moving coil is at the 'stop' position. During the 'keeper' phase, the meniscus at the nozzle tip is approximately a concave hemisphere. The surface tension exerts a net force on the ink, which results in the nozzle refilling, replacing the volume of the moving coil withdrawal with ink. This process takes approximately 100 μs.

The coil current is then turned off. The plunger is accelerated to its nominal position by the stressed spring. The force on the plunger is greatest at the beginning of the stroke, and slows as the spring elastic stress falls to zero. As a result, the acceleration is high at the beginning of the stroke, and the ink velocity is much more uniform during the stroke. The moving coil is then at the quiescent position until the next drop ejection cycle.

The drop firing rate is around 7 KHz. The ink jet head is suitable for fabrication as a monolithic pagewide print head. The print head has a 'roof shooter' configuration.

## IJ12 VARIABLE RELUCTANCE LINEAR STEPPER ACTUATOR INK JET

Ink Jet IJ12 is described with reference to the accompanying drawings in which:

Fig. C12.1 illustrates a single nozzle and actuator of a 1600 dpi ink jet. The moving pole is sandwiched between 12 coils which are driven in three phases. The order of the three phases determines whether the moving pole pushes the piston into the nozzle chamber (at rear) or withdraws it from the chamber.

Fig. C12.2 illustrates an exploded view of the variable reluctance linear stepper actuator, the piston, the nozzle chamber, and the nozzle. The nozzle radius is 8μ m for 1600 dpi half tone operation. Ink ingress to the nozzle chamber is via the integrated filter in the wall of the nozzle chamber.

IJ12 is a an ink jet head utilizing one miniature linear magnetic stepper actuator for each nozzle. The actuators are bulk fabricated on the wafer surface using VLSI processing techniques.

The general class of linear magnetic actuators include the Linear Induction Actuator (LIA), Linear Permanent Magnet Synchronous Actuator (LPMSA), Linear Reluctance Synchronous Actuator (LRSA), Linear Switched Reluctance Actuator (LSRA), and the Linear Stepper Actuator (LSA). All of these types of actuator can be adapted to operate as an ink jet actuator. The LSA is particularly suitable as it is driven by digital signals rather than phased sinusoidal signals.

The LSA is a relatively complex way of fabricating an ink jet actuator, and is also not as area or power efficient as some other types of actuator. However, it does have the advantage that the travel of the piston can be accurately controlled at multiple points. This makes it particularly suitable for contone operation.

In the example shown, the LSA is a double sided flat variable reluctance linear stepper actuator. It contains twelve solenoids, wired in series in three separate phases. The central movable core of soft magnetic material (e.g. NiFe) has a number of 'teeth' whose spacing is harmonically related to the spacing of the solenoid poles. These teeth result in a variable reluctance in the magnetic circuit. When the three sets of solenoids are sequentially energized in the order 1,2,3,1,2,3 the moving pole is 'stepped' towards the nozzle chamber, where it pushes a piston into the chamber. As the piston moves into the chamber, it ejects an ink drop. The piston is withdrawn from the chamber by energizing the solenoids in the order 3,2,1,3,2,1. The rate of movement of the piston in both directions can be finely controlled by varying the times that the solenoids are energized.

The piston is hydrophobic (e.g. PTFE), so ink does not flow out of the nozzle by capillarity between the piston and the nozzle chamber wall. Due to the hydrophobicity, the piston becomes suspended

in the middle of the ink meniscus, eliminating mechanical friction with the side walls of the nozzle chamber.

The drop firing rate is around 7 KHz. This is limited by the nozzle refill time. Nozzle refill requires approximately 100 μs, as it occurs as a result of surface tension. The print head has a 'roof shooter' configuration.

## IJ13 GEAR DRIVEN SHUTTER INK JET

Ink Jet IJ13 is described with reference to the accompanying drawings in which:

> Fig. C13.1 illustrates a single nozzle and actuators of a 1600 dpi print head viewed from the inside of the ink reservoir. The ink pressure in the reservoir oscillates with sufficient amplitude to eject drops when the shutter is open. The actuators are in a external magnetic field, which is normal to the plane of the device. Opposing pairs of actuators are pulsed 144 times to turn the gears and open or close the shutter.
> Fig. C13.2 illustrates a cross section of the nozzle, showing the gear construction.

IJ13 uses gears to allow a high speed actuator which produces a tiny force and a 1 micron travel to move a shutter across a nozzle chamber.
In the example shown, two actuators are each pulsed 144 times to move the shutter from an open to a closed condition. A second pair of actuators are each pulsed 144 times to move the shutter back.

The four actuators are each a single turn of wire which are immersed in a constant magnetic field, with the field being normal to the plane of the actuators. When a current is passed through each wire, the Lorenz force acting on the wire bends it a short distance. The wire incorporates two concertina springs, so the wire bends rather than stretches.

The gears are fabricated using standard MEMS processes, and may be made from two layers of polysilicon separated by two layers of sacrificial oxide. If fabricated from polysilicon, this must be done before the CMOS processing is complete, due to the high temperatures required for depositing and annealing the poly.

However, to minimize chip area it is desirable to fabricate the CMOS drive circuitry underneath the gear region. In this case, the gears and actuators may be fabricated using a low temperature process. One such method has been developed by Zavracky, McGruer and Morrison at Northeastern University, and uses electroplated nickel with a maximum processing temperature of 250 °C. This technique is also advantageous in that the CMOS processing flow does not need to be interrupted, so a standard CMOS process can be used.

The surface of the wafer is directly immersed in the ink reservoir, or in relatively large ink channels. An ultrasonic transducer (for example, a piezoelectric transducer) is positioned in the reservoir. The transducer oscillates the ink pressure at approximately 100 KHz. The ink pressure oscillation is sufficient that ink drops would be ejected from the nozzle were it not blocked by the shutter.

When data signals distributed on the print head indicate that a particular nozzle is to eject a drop of ink, the drive transistor pair of actuators which open the shutter are each pulsed 144 times at around 28 MHz. The peak of the ink pressure variation causes the ink to be squirted out of the nozzle. As the ink pressure goes negative, ink is drawn back into the nozzle, causing drop break-off. The shutter is kept open until the nozzle is refilled on the next positive pressure cycle. It is then shut by a series of 144 pulses to the opposite pair of actuators, to prevent the ink from being withdrawn from the nozzle on the next negative pressure cycle.

Each drop ejection takes two ink pressure cycles, resulting in a 50 KHz drop firing rate.

The nozzle chamber is formed using an anisotropic crystallographic etch of the silicon substrate. The device is manufactured on <100> silicon with a buried boron etch stop layer.

## IJ14 VARIABLE RELUCTANCE TAPERED MAGNETIC POLE INK JET

Ink Jet IJ14 is described with reference to the accompanying drawings in which:

Fig. C14.1 illustrates a cross section of a single nozzle and actuator of a 1600 dpi print head. The top includes a two layer pre-stressed nitride spring, the next layer is NiFe, the next is copper. The curved underside to the piston provides a continuously variable reluctance which increases the piston travel.
Fig. C14.2 illustrates an exploded view of the nozzle. Layers from the top are: a) low stress nitride, b) high stress nitride, c) NiFe, d) copper, e) NiFe, f) nitride passivation layer, g) oxide layer containing CMOS metallization, h) silicon layer containing CMOS transistors and the nozzle chamber, i) boron doped silicon etch stop layer containing the nozzle tip.

IJ14 is a direct firing electromagnetic ink jet. A solenoid attracts a soft magnetic piston which ejects the ink. The piston has a tapered inside thickness, providing a variable reluctance as the piston is drawn into the ink chamber.

The use of a tapered magnetic pole allows the travel of the piston to be much greater than a flat moving pole. The taper effectively 'smoothes out' the reluctance change in relation to gap width. When the piston is maximally withdrawn from the chamber, there is still some piston material in close proximity to the fixed pole. Thereby, the force applied to the piston is substantially more linear with distance.
The piston is held in position by a pre-stressed spring fabricated from two layers of silicon nitride of different stoichiometry. The bottom layer has more tension than the top layer, causing the spring to curl upwards into a cup shape when released by etching the layers of sacrificial glass used in the fabrication of the nozzle.

When the drive transistor for a nozzle is turned on, the solenoid surrounding the piston is energized, attracting the piston into the chamber. The piston pushes against the ink, causing ink to be squirted out of the nozzle. After approximately 2 μs, the current to the solenoid is turned off. At the same time, or at a slightly later time, a reverse current is applied, approximately half of the forward current. As the piston will carry some residual magnetism, this causes the piston to move backwards towards its nominal position. The bi-layer spring also helps to return the piston. The reverse current is turned off before the magnetization of the piston is reversed. The return of the piston to its quiescent position causes a low pressure in the ink chamber. This causes ink to begin flowing from the ejected drop back into the nozzle, and also ingests air into the chamber. The forward velocity of the drop and backward velocity of the ink in the chamber are resolved by the ink drop breaking off from the ink in the nozzle. The ink drop then continues to travel towards the recording medium under its own momentum. The nozzle refills due to the surface tension of the ink at the nozzle tip. Shortly after the time of drop breakoff, the meniscus at the nozzle tip is approximately a concave hemisphere. The surface tension exerts a net forward force on the ink, which results in the nozzle refilling.

The repetition rate of the ink jet is principally determined by the nozzle refill time, which will be approximately 100 μs, depending upon device geometry, ink surface tension, and the volume of the ejected drop.

## IJ15 LINEAR SPRING ELECTROMAGNETIC GRILL INK JET

Ink Jet IJ15 is described with reference to the accompanying drawings in which:

Fig. C15.1 illustrates a single nozzle and actuator of a 1600 dpi ink jet viewed from the inside of the ink reservoir. The nozzle actuator is in the quiescent position, with the shutter grill blocking ink access to the nozzle chamber.
Fig. C15.2 illustrates that when energized, the electromagnets attract the bar, causing the linear spring to move the grill sideways, exposing the nozzle chamber.
Fig. C15.3 illustrates an exploded view of the nozzle.

IJ15 is a magnetically actuated ink jet which uses a linear spring to increase the travel of a shutter grill.

An oscillating ink pressure is used to eject ink from the nozzles. Each nozzle has an associated shutter grill, which normally blocks the slots in a fixed grill over the nozzle chamber. The shutter grill is moved so as not to block the fixed grill slots by the electromagnetic actuator whenever an ink drop is to be fired.

The electromagnetic actuator attracts a soft magnetic (NiFe) bar, which is attached to the shutter grill. The bar is also connected in a simple linear spring arrangement to an anchor. The linear spring increases the movement of the shutter by a factor of eight. A one micron motion of the bar towards the electromagnets will result in an eight micron sideways movement. This dramatically improves the efficiency of the system, as the magnetic field falls off strongly with distance, while the spring has a linear relationship between motion in one axis and the other.

The surface of the wafer is directly immersed in the ink reservoir, or in relatively large ink channels. An ultrasonic transducer (for example, a piezoelectric transducer) is positioned in the reservoir. The transducer oscillates the ink pressure at approximately 100 KHz. The ink pressure oscillation is sufficient that ink drops would be ejected from the nozzle were it not blocked by the shutter.

When data signals distributed on the print head indicate that a particular nozzle is to eject a drop of ink, the drive transistor for that nozzle is turned on. This energizes the actuator, which moves the shutter so that it is not blocking the ink chamber. The peak of the ink pressure variation causes the ink to be squirted out of the nozzle. As the ink pressure goes negative, ink is drawn back into the nozzle, causing drop break-off. The shutter is kept open until the nozzle is refilled on the next positive pressure cycle. It is then shut to prevent the ink from being withdrawn from the nozzle on the next negative pressure cycle. Each drop ejection takes two ink pressure cycles, resulting in a 50 KHz drop firing rate

The nozzle chamber is formed using an anisotropic crystallographic etch of the silicon substrate. Etchant access to the substrate is via the slots in the fixed grill. The device is manufactured on <100> silicon (with a buried boron etch stop layer), but rotated 45° in relation to the <010> and <001> planes. Therefore, the <111> planes which stop the crystallographic etch of the nozzle chamber form a 45° rectangle which superscribes the slots in the fixed grill.

## IJ16 LORENZ DIAPHRAGM ELECTROMAGNETIC INK JET

Ink Jet IJ16 is described with reference to the accompanying drawings in which:

Fig. C16.1 illustrates a single nozzle and actuator of a 1600 dpi print head viewed from the inside of the ink reservoir.
Fig. C16.2 illustrates a cross section of the nozzle showing drop ejection.

IJ16 uses the Lorenz force on a current carrying wire in a magnetic field. The static magnetic field is provided by a permanent magnet yoke around the ink jet head.

The actuator consists of a planar copper coil. A section of the coil is embedded in a corrugated diaphragm, which is suspended over a nozzle chamber. The electric current in the diaphragm coil section all flows in one direction. The external magnetic field is in the plane of the chip surface, perpendicular to the current flow in the diaphragm coil. The Lorenz interaction of the diaphragm coil current and the magnetic field results in a force which pushes the diaphragm towards the nozzle, ejecting a drop of ink. The diaphragm is corrugated so that flexure occurs as an elastic bending motion. This is essential, as tensile stress would prevent a flat diaphragm from flexing.

After approximately 3 μs, the coil current is turned off, and the diaphragm returns to its quiescent position. The diaphragm return 'sucks' some of the ink back into the nozzle, causing the ink ligament connecting the ink drop to the ink in the nozzle to thin. The forward velocity of the drop and backward velocity of the ink in the nozzle cause the ink drop to break off from nozzle ink. The ink drop then continues towards the recording medium. Ink refill of the nozzle chamber is via the two slots at either side of the diaphragm. The ink refill is caused by the surface tension of the ink meniscus at the nozzle, and takes around 100 μs..

The diaphragm corrugations are formed by depositing resist over a layer of sacrificial glass. The resist is exposed using a halftone mask delineating the corrugation lines. After development, the resist contains thickness corrugations. The resist and sacrificial glass are etched using an etchant which erodes the resist at substantially the same rate as the glass. This transfers the corrugated thickness pattern onto the sacrificial glass. A nitride passivation layer is deposited on the glass. This is followed by a copper

layer, which is patterned using a coil mask. A further nitride passivation layer follows. The slots in the nitride layer at the side of the diaphragm are then etched. This is followed by etching the sacrificial glass.

The nozzle chamber is formed using an anisotropic crystallographic etch of the silicon substrate. Etchant access to the substrate is via the slots at the sides of the diaphragm. The device is manufactured on <100> silicon (with a buried boron etch stop layer), but rotated 45° in relation to the <010> and <001> planes. Therefore, the <111> planes which stop the crystallographic etch of the nozzle chamber form a 45° rectangle which superscribes the slots in the nitride layer. This etch will proceed quite slowly, due to limited access of etchant to the silicon. However, the etch can be performed at the same time as the bulk silicon etch which thins the wafer.

The drop firing rate is around 7 KHz. The print head has a 'down shooter' configuration.

## IJ17 PTFE SURFACE SHOOTING SHUTTERED INK JET

Ink Jet IJ17 is described with reference to the accompanying drawings in which:

Fig. C17.1 illustrates a single nozzle and actuator of a 1600 dpi print head.
Fig. C17.2 illustrates an exploded view of the nozzle, shutter, and actuator.

IJ17 uses an oscillating ink pressure to eject ink from nozzles. Each nozzle has an associated shutter, which normally blocks it. The shutter is moved away from the nozzle opening by thermal bend actuator whenever an ink drop is to be fired.

The nozzles are connected to ink chambers which contain the actuators. These chambers are connected to ink supply channels which are etched through the silicon wafer. The ink supply channels are substantially wider then the nozzles, to reduce the fluidic resistance to the ink pressure wave. The ink channels are connected to an ink reservoir. An ultrasonic transducer (for example, a piezoelectric transducer) is positioned in the reservoir. The transducer oscillates the ink pressure at approximately 100 KHz. The ink pressure oscillation is sufficient that ink drops would be ejected from the nozzle were it not blocked by the shutter.

The shutters are moved by a thermoelastic actuator. The actuators are formed as a coiled serpentine copper heater embedded in polytetrafluoroethylene (PTFE). PTFE has a very high coefficient of thermal expansion (approximately 770 x $10^{-6}$). The current return trace from the heater is also embedded in the PTFE actuator. The current return trace is made wider than the heater trace, and is not serpentine. Therefore, it does not heat the PTFE as much as the serpentine heater does. The serpentine heater is positioned along the inside edge of the PTFE coil, and the return trace is positioned on the outside edge. When actuated, the inside edge becomes hotter than the outside edge, and expands more. This results in the actuator uncoiling.

The heater layer is etched in a serpentine manner both to increase its resistance, and to reduce its effective tensile strength along the length of the actuator. This is so that the low thermal expansion of the copper does not prevent the actuator from expanding in response to the high thermal expansion of the PTFE.

When data signals distributed on the print head indicate that a particular nozzle is to eject a drop of ink, the drive transistor for that nozzle is turned on. This energizes the actuator, which moves the shutter so that it is not blocking the ink chamber. The peak of the ink pressure variation causes the ink to be squirted out of the nozzle. As the ink pressure goes negative, ink is drawn back into the nozzle, causing drop break-off. The shutter is kept open until the nozzle is refilled on the next positive pressure cycle. It is then shut to prevent the ink from being withdrawn from the nozzle on the next negative pressure cycle.

Each drop ejection takes two ink pressure cycles. Half of the nozzles should eject drops in one phase, and the other half of the nozzle should eject drops in the other phase. This minimizes the pressure variations which occur due to different numbers of nozzles being actuated.

The amplitude of the ultrasonic transducer can be altered in response to the viscosity of the ink (which is typically affected by temperature), and the number of drops which are to be ejected in the current cycle. This amplitude adjustment can be used to maintain consistent drop size in varying environmental conditions.

The drop firing rate is around 50 KHz. The ink jet head is suitable for fabrication as a monolithic pagewide print head. The print head has an 'up shooter' configuration.

## IJ18 PUSH-PULL BUCKLE STRIP GRILL INK JET

Ink Jet IJ18 is described with reference to the accompanying drawings in which:

Fig. C18.1 illustrates a single nozzle and actuator of a 1600 dpi print head.
Fig. C18.2 illustrates an exploded view of the nozzle, shutter, and actuator.

IJ18 is a fast, extremely low energy ink jet system which uses an oscillating ink pressure to eject ink from nozzles. Each nozzle has an associated fixed grill and moving grill. The slots in the fixed and the bars of the moving grill are normally aligned. This blocks ink flow into the nozzle chamber. When the electrothermal bend actuators are energized, the moving grill shifts so that its slots line up with the slots of the fixed grill. When this occurs ink flow in and out of the nozzle chamber is relatively unobscured.

The grill is moved by a pair of thermoelastic actuators. One of the actuators pushes the grill, and the other pulls it. The actuators contain serpentine metal wires which are mechanically and electrically connected to the substrate at both ends. The four point mechanical connection gives the grill a high degree of stability and rigidity. The serpentine copper heaters of the actuators are embedded in polytetrafluoroethylene (PTFE). PTFE has a very high coefficient of thermal expansion (approximately $770 \times 10^{-6}$). The serpentine heater is positioned along one edge of the PTFE actuator. When actuated, this edge becomes hotter than the opposite edge, and expands more. This results in the actuator bending. The two actuators are arranged so that both bend in the same direction. The movable grill is fabricated from the same PTFE layer as the actuators, and is integrally connected between the centers of the two actuators. The heater layer is etched in a serpentine manner both to increase its resistance, and to reduce its effective tensile strength along the length of the actuator. This is so that the low thermal expansion of the copper does not prevent the actuator from expanding in response to the high thermal expansion of the PTFE. The PTFE is made hydrophilic so that the nozzle fills by capillarity.

The grills lie between ink channels which are etched through the silicon wafers, and nozzle chambers which are fabricated as hollow structures of silicon nitride. The ink supply channels are substantially wider then the nozzles, to reduce the fluidic resistance to the ink pressure wave. The ink channels are connected to an ink reservoir. An ultrasonic transducer (for example, a piezoelectric transducer) is positioned in the reservoir. The transducer oscillates the ink pressure at approximately 100 KHz. The ink pressure oscillation is sufficient that ink drops would be ejected from the nozzle were it not blocked by the shutter.

When a drop is to be printed, the actuators are energized, moving the shutter so that it does not block the ink chamber. The peak of the ink pressure variation causes the ink to be squirted out of the nozzle. As the ink pressure goes negative, ink is drawn back into the nozzle, causing drop break-off. The shutter is kept open until the nozzle is refilled on the next positive pressure cycle. It is then shut to prevent the ink from being withdrawn from the nozzle on the next negative pressure cycle.

The drop firing rate is around 50 KHz. The ink jet head is suitable for fabrication as a monolithic pagewide print head. The print head has an 'up shooter' configuration.

## IJ19 THERMOELASTIC BEND ACTUATOR SHUTTERED INK JET

Ink Jet IJ19 is described with reference to the accompanying drawings in which:

Fig. C19.1 illustrates a pair of nozzles and actuators of a 1200 dpi print head viewed from the inside of the ink reservoir. The closer nozzle is open, and the further nozzle is closed.
Fig. C19.2 illustrates an exploded view of the pair of nozzles.

IJ19 uses an oscillating ink pressure to eject ink from nozzles. Each nozzle has an associated shutter and fixed grill, made from polysilicon. There are two polysilicon thermoelastic bend actuators for each nozzle. One of these actuators is mechanically connected to the shutter, and moves it into the 'open' position whenever a drop is to be fired. The other actuator is much smaller, and forms a 'catch' to hold the shutter open during the drop ejection cycle. Energizing the catch actuator causes it to bend away from a notch formed in the shutter, releasing the shutter and allowing it to return to the closed position.

The hot arm of each bend actuator is coated with a polymer which has a poor thermal conductivity. The cold arm of each bend actuator is in thermal contact with the water based ink.

The nozzles are connected to ink chambers, which in turn are connected to an ink reservoir. An ultrasonic transducer (typically a piezo electric transducer) is positioned in the reservoir. The transducer oscillates the ink pressure at approximately 100 KHz. The ink pressure oscillation is such that ink drops would be ejected from the nozzle.

Ink drop ejection takes two cycles of the pressure wave. The shutter actuator is energized at the beginning of the positive pressure phase of the first cycle. The shutter moves into the open position, and the catch locks it in position. After a short period, the power to the shutter actuator is turned off, and the hot arm of the actuator begins to cool. Meanwhile, the high ink pressure forces a drop of ink from the nozzle. The ink pressure then goes into the negative pressure phase, whereupon ink around the base of the ejected drop is sucked back into the nozzle, along with an air bubble. This causes the ejected ink drop to break off from the ink in the nozzle. The shutter is kept open until the nozzle is refilled on the next positive pressure cycle. At the end of the refill half-cycle, the catch actuator is energized. This allows the shutter to rapidly return to its nominal position, preventing the ink from being withdrawn from the nozzle on the next negative pressure cycle.

The catch has two purposes: to reduce power consumption by holding the shutter open without requiring power to the main actuator, and to allow the main actuator time to cool down before it is to shut, thereby achieving a rapid closing of the shutter.

The amplitude of the ultrasonic transducer can be altered in response to the viscosity of the ink (which is typically affected by temperature), and the number of drops which are to be ejected in the current cycle.

The drop firing rate is around 50 KHz. The ink jet head is suitable for fabrication as a monolithic pagewide print head. The print head has a 'down shooter' configuration.

## IJ20 CURLING CALYX THERMOELASTIC INK JET

Ink Jet IJ20 is described with reference to the accompanying drawings in which:

Fig. C20.1 illustrates a cross section of a single nozzle showing 4 of the 8 actuator 'petals'.
Fig. C20.2 illustrates when the actuator is energized, the 'petals' curl into a calyx formation. Ink above the actuator is trapped by the nozzle chamber wall, and is expelled from the nozzle. Ink flows in under the nozzle chamber wall to fill the expanding region under the actuator.
Fig. C20.3 illustrates an exploded view of the nozzle.

IJ20 has one thermoelastic bend actuator for each nozzle. The center of the actuator is mechanically and electrically connected to the substrate. Around the center are eight petal shaped thermoelastic bend actuators. These 'petals' are wired in series, and are all energized simultaneously. Each 'petal' contains an electrothermal heater composed of copper or other electrically conducting material. The copper heater is embedded in a polytetrafluoroethylene (PTFE) paddle. PTFE is used because it has a very high coefficient of thermal expansion.

When data signals distributed on the print head indicate that a particular nozzle is to eject a drop of ink, the drive transistor for that nozzle is turned on for around 3 μs. This energizes the heaters in the 'petals' which heat the lower half of each petal. The top half is not heated, and is cooled by the water based ink. The lower PTFE expands rapidly. This expansion causes the petals to curl away from the substrate into a calyx formation, pushing ink out of the nozzle.

When the heater current is turned off, the petals return to their quiescent position. This 'sucks' some of the ink back into the nozzle, causing the ink drop to break off from the ink in the nozzle. The ink drop then continues towards the recording medium.

The nozzle is fabricated as a hole in an integrated nozzle plate of silicon nitride. The nozzle plate is raised from the substrate to form the nozzles and their chambers by the use of a sacrificial oxide. Before deposition of the nitride, the sacrificial oxide is etched to two different depths. A circular rim is etched to the depth of the actuator to form the suspended nozzle chamber. A series of holes are also

etched right through the sacrificial oxide. When filled with nitride these form posts which suspend the nozzle plate above the actuators.

The top surface of the PTFE is treated to make it hydrophilic. The lower surface can be left hydrophobic. In this case, an air bubble will form under the petals, as the nozzle will not completely fill by capillarity.

The air bubble prevents the water based ink contacting the underside of the paddle, achieving a higher temperature with lower power.

The drop firing rate is around 7 KHz, limited by the nozzle refill time. The ink jet head is suitable for fabrication as a monolithic pagewide print head.

## IJ21 SHUTTERED OSCILLATING PRESSURE INK JET

Ink Jet IJ21 is described with reference to the accompanying drawings in which:

Fig. C21.1 illustrates a single nozzle and actuator of a 1600 dpi print head viewed from the inside of the ink reservoir.
Fig. C21.2 illustrates an exploded view of the nozzle.

IJ21 uses an oscillating ink pressure to eject ink from nozzles. Each nozzle has an associated shutter, which is moved by an electrothermal bend actuator. The shutter is normally open, and is moved to cover the ink chamber whenever an ink drop is to be prevented from being ejected.

The nozzles are connected to ink chambers, which in turn are connected to an ink reservoir. An ultrasonic transducer (typically a piezo electric transducer) is positioned in the reservoir. The transducer oscillates the ink pressure at approximately 100 KHz. The ink pressure oscillation is sufficient to eject ink drops from the nozzle on every cycle if the shutter is open.

The coiled actuator is constructed from laminated conductors of either differing resistivities, different cross sectional areas, different indices of thermal expansion, different thermal conductivities to the ink, different lengths, or some combination thereof. The coiling radius of the actuator changes when a current is passed through it, as one side of the coiled beam expands differently than the other.

The actuator controls the position of the shutter, so that it can cover none, all or part of the nozzle chamber. If the shutter covers none of the nozzle chamber, then the oscillating ink pressure is transmitted to the nozzle. If the shutter covers the ink chamber, then the oscillating ink pressure is significantly attenuated at the nozzle. The ink pressure variations are not entirely stopped, due to leakage around the shutter, and flexing of the shutter under varying pressure. The shutter may also be part way across the ink chamber, resulting in partial attenuation of the ink pressure variations. This may be used to vary the volume of the ejected drop. Drop volume control may be used either to implement a degree of continuous tone operation, to regulate the drop volume, or both.

When data signals distributed on the print head indicate that a particular nozzle is to eject a drop of ink, the drive transistor for that nozzle is turned off. This de-energizes the actuator, allowing the shutter to move so that it is not blocking the ink chamber. The peak of the ink pressure variation causes the ink to be squirted out of the nozzle. As the ink pressure goes negative, ink is drawn back into the nozzle, causing drop break-off. The shutter is left open until the nozzle is refilled on the next positive pressure cycle. It is then shut to prevent the ink from being withdrawn from the nozzle on the next negative pressure cycle. Each drop ejection takes two ink pressure cycles.

The drop firing rate is around 50 KHz. The ink jet head is suitable for fabrication as a monolithic pagewide print head. The print head has a 'down shooter' configuration.

## IJ22 IRIS MOVEMENT THERMOELASTIC BEND ACTUATOR INK JET

Ink Jet IJ22 is described with reference to the accompanying drawings in which:

Fig. C22.1 illustrates a single nozzle and actuator of a 1600 dpi surface shooting print head.
Fig. C22.2 illustrates a view of the nozzle in quiescent position, with the nozzle plate removed.

Fig. C22.3 illustrates a view of the nozzle in energized position, with the nozzle plate removed.

IJ22 has four thermoelastic bend actuators for each nozzle. The actuators are energized simultaneously. Attached to each actuator is a curved upright vane. When energized, the actuators cause the vanes to contract in a manner similar to a camera iris. This contraction reduces the volume contained by the vanes, causing ink to be ejected from the nozzle. When power to the actuators is turned off, they cool down rapidly, returning to their normal position. As they return, the pressure in the space enclosed by the vanes falls, causing some of the ink to be 'sucked' back into the nozzle. This results in the ejected drop breaking off from the ink in the nozzle, and travelling towards the print medium.

The actuators each have two arms, an expanding, flexible arm, and a rigid arm. Both arms are fixed at one end, and are joined together at the other end. The thermally expanding arms are formed of a serpentine copper heater embedded in polytetrafluoroethylene (PTFE). The heater layer is etched in a serpentine manner both to increase its resistance, and to reduce its effective tensile strength along the length of the actuator. This is so that the low thermal expansion of the copper does not prevent the actuator from expanding in response to the high thermal expansion of the PTFE. The rigid arm comprises the return trace of the copper heater, and the vane. The vane is not present along the entire length of the rigid arm. Approximately 20% of the length of the rigid arm has no vane, and is flexible. This section of the rigid arm bends in response to the force created by the elongation of the expanding arm. This localization of the bending results in a greater overall movement of the vane.

Ink flow to the nozzle is across the chip surface. For low speed printers this can be supplied via the chip edge. For higher speed operation, ink channels should be etched through the substrate to increase the ink supply.

The drop firing rate is around 7 KHz. The printing speed is primarily constrained by the nozzle refill time, which is approximately 100 μs if the ink is not under pressure. The ink jet head is suitable for fabrication as a monolithic pagewide print head.

## IJ23 DOWN SHOOTER WITH PTFE THERMOELASTIC BEND ACTUATOR

Ink Jet IJ23 is described with reference to the accompanying drawings in which:

Fig. C23.1 illustrates a cross section of a single nozzle and actuator of a 1600 dpi print head, shown during the firing of a drop.
Fig. C23.2 illustrated an exploded view of the nozzle.

IJ23 has one thermoelastic bend actuator for each nozzle. One end of the actuator is mechanically connected to the substrate, and the other end is connected to a stiff paddle. When energized, the actuator bends into a nozzle chamber in the substrate. This motion causes the attached paddle to eject a drop of ink through a nozzle at the opposite side of the nozzle chamber. As the actuator is cooled by the ink, it returns to its nominal position.

The actuator is composed of a copper heater embedded in a polytetrafluoroethylene (PTFE) paddle. PTFE is used because it has a very high coefficient of thermal expansion.

The bottom layer of PTFE is thick (around 3 μm) and is not significantly heated by the heater in the brief time that the heater is energized. The copper heater is formed as a serpentine wire of approximately 0.3 μm thickness. The copper is deposited over thickness corrugations in the bottom PTFE layer, and then etched in the serpentine pattern. The PTFE corrugations can be formed by exposing a resist layer to a halftone mask, then etching the resist and the underlying PTFE at the same rate. This is done to increase the response speed of the actuator, by reducing the thermal distance between the heater and the PTFE that is to be heated. After heater deposition and etching, a further 0.5 μm of PTFE is deposited to make the hot side of the bend actuator. The PTFE is treated to make it hydrophilic.

When data signals distributed on the print head indicate that a particular nozzle is to eject a drop of ink, the drive transistor for that nozzle is turned on. This energizes the heater in the paddle for that nozzle. The heater is energized for approximately 3 μs, with the actual duration depending upon the design chosen for the actuator and nozzle. The heater heats the upper PTFE layer, but does not have time to significantly heat the lower thick PTFE layer. This expansion causes the paddle to bend, pushing ink out of the nozzle.

When the heater current is turned off, the paddle begins to return to its quiescent position. The paddle return 'sucks' some of the ink back into the nozzle, causing the ink drop to break off from the ink in the nozzle. The ink drop then continues towards the recording medium.

The drop firing rate is around 7 KHz. The ink jet head is suitable for fabrication as a monolithic pagewide print head. The print head has a 'down shooter' configuration.

## IJ24 CONDUCTIVE PTFE THERMOELASTIC BEND ACTUATOR INK JET

Ink Jet IJ24 is described with reference to the accompanying drawings in which:

Fig. C24.1 illustrates a single nozzle and actuator of a 1600 dpi print head.
Fig. C24.2 illustrates an exploded view of the nozzle.
Fig. C24.3 illustrates two nozzle chambers, showing the air vent between them.

IJ24 has one thermoelastic bend actuator for each nozzle. The bend actuator is constructed from a dual layer of PTFE. The bottom layer is doped to become conductive, forming an electrothermal heater. One end of the actuator is mechanically connected to the substrate, and the other end is connected to a stiff paddle. When energized, the actuator bends away from the substrate towards a nozzle, causing the attached paddle to eject ink. As the actuator is cooled by the ink, it returns to its nominal position. The nozzle is fabricated as a hole in an integrated nozzle plate of silicon nitride. The nozzle plate is raised from the substrate to form the ink channels, nozzles, filters, and nozzle chambers by the use of a sacrificial oxide.

The actuator is composed of two layers:

1) A conductive polytetrafluoroethylene (PTFE) lower layer, of around 1 μm thickness. PTFE has a very high coefficient of thermal expansion (approximately 770 x $10^{-6}$, or around 380 times that of silicon). The PTFE is made conductive by dispersing conductive material in the polymer matrix. For example, a dispersion of about 2% of single wall carbon nanotubes of average length of about 1 μm. The resistivity required for convenient low voltage operation is around 100 μΩm.
2) A PTFE upper layer of around 3.5 μm thickness. The upper surface of the PTFE is treated to make it hydrophilic. This may be achieved by exposing the PTFE surface to a high energy plasma in high vacuum, with ammonia inlet. The plasma breaks the surface PTFE chains, the ends of which are then available to react with the ammonia, forming partial positive charges on terminal $NH_2$ groups, and negative charges on the fluorine.

When a drop is to be fired, the drive transistor for that nozzle is turned on for around 4 μs. This energizes the lower (conductive) PTFE, wherein electrical resistive losses are converted to heat. The top PTFE is not heated, and is cooled by the water based ink. The lower PTFE expands rapidly. This expansion causes the actuator to bend, moving the paddle which pushes ink out of the nozzle. There is an air bubble between the paddle and the substrate, which forms due to the hydrophobic nature of the PTFE on the back surface of the paddle. An air vent connects the air bubble to the ambient air, allowing the paddle to move without being held back by a large pressure drop under the paddle. The air bubble also prevents the water based ink contacting the underside of the paddle, achieving a higher temperature with lower power.

When the heater current is turned off, the paddle begins to return to its quiescent position. The paddle return causes the ink drop to break off from the ink in the nozzle. The ink drop then continues towards the recording medium.

## IJ25 MAGNETOSTRICTIVE INK JET

Ink Jet IJ25 is described with reference to the accompanying drawings in which:

Fig. C25.1 illustrates a view of a single 1600 dpi actuator. The magnetostrictive material (Terfenol-D) is surrounded by a copper coil. Below the magnetostrictive paddle is the nozzle chamber, which includes the nozzle.
Fig. C25.2 illustrates a cross section of the nozzle showing the copper coil, the Terfenol-D layer which forms the bend actuator of the paddle, a nitride layer which provides a

bend 'foil' of high tensile strength, a crystallographically etched nozzle chamber in silicon, and a nozzle membrane fabricated in boron doped silicon.

IJ25 uses bend actuator which operates on magnetostrictive principles. The actuator uses the giant magnetostrictive effect of materials such as Terfenol-D (an alloy of terbium, dysprosium and iron developed at the Naval Ordnance Laboratory, hence Ter-Fe-NOL).

The actuator is suspended at the entrance to a nozzle chamber, on the opposite side to the nozzle. The actuator is in the center of a coil of copper, which forms a solenoid. Copper is chosen for the solenoid both for its low resistivity and its high resistance to electromigration. The copper coil is directly cooled by the ink, so relatively large electrical currents can be accommodated without a significant temperature rise. When the solenoid is energized, the resultant magnetic field causes the Terfenol-D layer of the actuator to expand. the actuator bends into the nozzle chamber, and pushes ink out of the nozzle.

There is one magnetostrictive bend actuator for each nozzle. One end of the bend actuator is mechanically connected to the substrate. The other end is free to move.

The actuator is composed of three layers:

1) An silicon nitride bottom layer. This layer is of high stiffness, which is deposited stress free with the appropriate stoichiometry. Its purpose is to prevent the actuator from elongating, so that the expansion of the magnetostrictive layer is manifested as a bending motion.
2) A magnetostrictive layer. A material with giant magnetostrictive properties such as Terfenol-D is used. The layer thickness is chosen to have approximately the same tensile strength as the thick nitride layer.
3) A thin silicon nitride passivation layer, to prevent corrosion of the Terfenol-D. This is actually made of two layers: the nitride insulation between the Terfenol-D and the coil, and the coil passivation layer. The total thickness should be less than a third of the thickness of the bottom nitride layer.

The drop firing rate is around 7 KHz, constrained by the nozzle refill time. The ink jet head is suitable for fabrication as a monolithic pagewide print head, which has a 'down shooter' configuration.

## IJ26 SHAPE MEMORY ALLOY INK JET

Ink Jet IJ26 is described with reference to the accompanying drawings in which:

Fig. C26.1 illustrates a single nozzle and actuator of a 1600 dpi print head in the quiescent position. The nozzle is viewed from the inside of the ink reservoir.
Fig. C26.2 illustrates a cross section of the nozzle firing a drop. The SMA actuator is in its austenic phase due to being electrothermally heated.
Fig. C26.3 illustrates how the SMA actuator returns to its martensitic state as it cools down. Elastic stress in the nitride layer returns it to a bent shape.

IJ26 relies upon the thermal transition of a shape memory alloy (SMA) from its martensitic phase to its austenic phase. The thermal transition is achieved by passing an electrical current through the SMA. The actuator is suspended at the entrance to a nozzle chamber, on the opposite side to the nozzle. The actuator is bent away from the nozzle when in its quiescent state. When energized, the actuator straightens, and pushes ink out of the nozzle. Energizing the actuator requires supplying enough energy to raise the SMA above the transition temperature, and to provide the latent heat of transformation to the SMA.

The SMA martensitic phase must be pre-stressed to achieve a different shape from the austenic phase. For print heads with many thousands of nozzles, it is important to achieve this re-stressing in a bulk manner. IJ26 achieves this by depositing a layer of silicon nitride using PECVD at around 300 °C over the SMA. This deposition occurs while the SMA is in the austenic shape. After the print head cools to room temperature, the substrate under the SMA bend actuator is removed by chemical etching. The silicon nitride is under tensile stress, and causes the actuator to curl upwards. The weak martensitic phase of the SMA provides little resistance to this curl. When the SMA is heated to its austenic phase, it returns to the flat shape into which it was annealed during the nitride deposition.

There is one SMA bend actuator for each nozzle. One end of the SMA bend actuator is mechanically connected to the substrate. The other end is free to move under the stresses inherent in the layers.

The nozzle plate is formed from a buried etch stop layer in the silicon substrate.

The actuator is composed of three layers:

1) An $SiO_2$ lower layer. This layer acts as a stress 'reference' for the nitride tensile layer. It also protects the SMA from the crystallographic silicon etch that forms the nozzle chamber. This layer is formed as part of the standard CMOS process for the active electronics of the print head.
2) A SMA heater layer. A SMA such as nickel titanium (NiTi) alloy is deposited and etched into a serpentine form to increase the electrical resistance.
3) An silicon nitride top layer. This is a thin layer is of high stiffness which is deposited using PECVD The nitride stoichiometry is adjusted to achieve a layer with significant tensile stress at room temperature relative to the $SiO_2$ lower layer. Its purpose is bend the actuator at the low temperature martensitic phase.

The drop firing rate is around 7 KHz. The ink jet head is suitable for fabrication as a monolithic pagewide print head. The print head has a 'down shooter' configuration.

## IJ27 THERMOELASTIC BUCKLE PLATE SURFACE SHOOTER INK JET

Ink Jet IJ27 is described with reference to the accompanying drawings in which:

Fig. C27.1 illustrates a single nozzle and actuator of a 1600 dpi print head.
Fig. C27.2 illustrates an exploded view of the nozzle. Two layers of PTFE are shown. The top layer is the buckle plate actuator, and the bottom layer is a hydrophobic vent grill which allows air into the space under the buckle plate.

IJ27 has one thermoelastic buckle plate actuator for each nozzle. Both ends of the actuator is mechanically connected to the substrate. When energized, a heater embedded in the actuator causes the actuator to expand. As both ends are fixed, the expansion causes a buckling in the center. To ensure that the buckling always occurs upwards, the heater is positioned closer to the top surface of the actuator than the bottom surface. As the buckle plate bends upwards, it pushes ink out of the nozzle. The power to the actuator is maintained for approximately 3 μs.

There is an air bubble between the buckle plate and the substrate, which forms due to the hydrophobic nature of the PTFE on the back surface of the buckle plate. An air vent connects the air bubble to the ambient air, allowing the buckle plate to move without being held back by a reduction in air pressure as the bubble expands. The air bubble also prevents the water based ink contacting the underside of the buckle plate, achieving a higher temperature with lower power.

When the heater current is turned off, the actuator is rapidly cooled by the ink. The buckle plate begins to return to its quiescent position. The buckle plate return 'sucks' some of the ink back into the nozzle, causing the ink drop to break off from the ink in the nozzle. The ink drop then continues towards the recording medium. The return of the buckle plate forces the air under the buckle plate back out of the vents.

The nozzle is fabricated as a hole in an integrated nozzle plate of silicon nitride. The nozzle plate is raised from the substrate to form the ink channels, nozzles, filters, and nozzle chambers by the use of a sacrificial oxide.

The actuator is composed of a copper heater embedded in a polytetrafluoroethylene (PTFE) buckle plate. PTFE is used because it has a very high coefficient of thermal expansion. The copper heater is formed as a serpentine wire of approximately 0.3 μm thickness. The upper surface of the PTFE is treated to make it hydrophilic.

The drop firing rate is around 7 KHz. The ink jet head is suitable for fabrication as a monolithic pagewide print head.

## IJ28 THERMOELASTIC ROTARY IMPELLER SURFACE SHOOTING INK JET

Ink Jet IJ28 is described with reference to the accompanying drawings in which:

Fig. C28.1 illustrates a single nozzle and actuator of a 1600 dpi surface shooting print head.
Fig. C28.2 illustrates an array of nozzles for a single ink color. The spacing between nozzles is 64 μ m,so four rows are required for 1600 dpi operation (with 16u mdot spacing). The holes between the nozzle rows are ink channels, etched completely through the wafer. Ink is supplied to the print head at the back of the wafer.

IJ28 has one rotary impeller for each nozzle. The rotary impeller has a set of moving vanes and a set of fixed vanes. When actuated, the set of moving vanes rotates towards the fixed vanes, displacing the ink between the vanes. The vanes are arranged so that the path of least resistance for the displaced ink is out of the nozzle.

The fixed vanes of the impeller are connected to the nozzle plate, which is suspended above the substrate on fixed posts. The moving vanes are attached to a copper rotating frame. One end of each of two thermoelastic expansion actuators are attached to the rotating frame. The actuators are connected to the rotating frame tangentially along the outer rim of the frame, in opposite directions.

The actuators are formed of a serpentine copper heater embedded in polytetrafluoroethylene (PTFE). The heater layer is etched in a serpentine manner both to increase its resistance, and to reduce its effective tensile strength along the length of the actuator. This is so that the low thermal expansion of the copper does not prevent the actuator from expanding in response to the high thermal expansion of the PTFE.

The copper heater forms a continuous circuit through both actuators, via the rotating frame. The other ends of the actuators are mechanically connected to the substrate, and electrically connected to the drive circuitry.

When data signals distributed on the print head indicate that a particular nozzle is to eject a drop of ink, the drive transistor for that nozzle is turned on. Both actuators are energized simultaneously.

The heater heats the PTFE layer, which expands. This expansion causes the actuators to elongate. The two actuators expand in opposite directions, and are connected to opposite sides of the rotating frame. The resultant forces on the frame cause a rotation of approximately 30 degrees.

When the heater current is turned off, the actuators return to their quiescent positions, rotating the frame back to its normal position.

The return of the frame rotates the moving vanes away from the fixed vanes. This 'sucks' some of the ink back into the nozzle, causing the ink ligament connecting the ink drop to the ink in the nozzle to thin. The forward velocity of the drop and backward velocity of the ink in the nozzle are resolved by the ink drop breaking off from the ink in the nozzle. The ink drop then continues towards the recording medium.

The actuator and impeller are at the quiescent position until the next drop ejection cycle.

The drop firing rate is around 7 KHz. The ink jet head is suitable for fabrication as a monolithic pagewide print head. The print head has a 'roof shooter' configuration.

## IJ29 PTFE-BASED THERMOELASTIC BEND ACTUATOR INK JET

Ink Jet IJ29 is described with reference to the accompanying drawings in which:

Fig. C29.1 illustrates a single nozzle and actuator of a 1600 dpi print head.
Fig. C29.2 illustrates an array of nozzles for a single ink color. The spacing between nozzles is 32 μ m.so two rows are required for 1600 dpi operation (with 16μm dot spacing). The holes between the nozzle rows are ink channels, etched completely through the wafer.

IJ29 has one thermoelastic bend actuator for each nozzle. One end of the actuator is mechanically connected to the substrate, and the other end is connected to a stiff paddle. When energized, the actuator bends away from the substrate towards a nozzle, causing the attached paddle to eject ink. As the actuator is cooled by the ink, it returns to its nominal position. The nozzle is fabricated as a hole in an integrated nozzle plate of silicon nitride. The nozzle plate is raised from the substrate to form the ink channels.

nozzles, filters, and nozzle chambers by the use of a sacrificial oxide. This is to avoid the requirement for separate fabrication and high precision assembly of a substrate and a nozzle plate.

The actuator contains four layers:

1) A polytetrafluoroethylene (PTFE) lower layer. PTFE has a very high coefficient of thermal expansion (approximately 770 x $10^{-6}$, or around 380 times that of silicon).
2) A heater layer. A serpentine heater is etched in this layer, which may be Nichrome, copper or other suitable material with a resistivity such that the drive voltage for the heater is compatible with the drive transistors. The heater is formed in serpentine fashion so as to have very little tensile strength in the direction along the length of the actuator.
3) A PTFE upper layer. This layer also expands when heated by the heater.
4) A silicon nitride layer. This is a thin layer is of high stiffness and low coefficient of thermal expansion. Its purpose is to ensure that the actuator bends, instead of simply elongating. Silicon nitride is used because it is a standard semiconductor material, and $SiO_2$ cannot easily be used if it is also the sacrificial material.

When a drop is to be fired, the drive transistor is turned on, energizing the heater for around 3 µs. The heater heats the PTFE layer, which expands rapidly. This expansion causes the actuator to bend, moving the paddle, and thereby pushing ink out of the nozzle. There is an air bubble between the paddle and the substrate, which forms due to the hydrophobic nature of the PTFE on the back surface of the paddle. This air bubble reduces the thermal coupling to the hot side of the actuator, achieving a higher temperature with lower power. The cold side of the actuator is cooled by the water based ink. The presence of the air bubble also means that less ink is required to move under the paddle when the actuator is energized. These factors lead to a lower power consumption of the actuator.

When the heater current is turned off, the paddle begins to return to its quiescent position. The paddle return 'sucks' some of the ink back into the nozzle, causing the ink drop to break off from the ink in the nozzle.

The drop firing rate is around 7 KHz.

## IJ30 PTFE AND CORRUGATED COPPER THERMOELASTIC BEND ACTUATOR

Ink Jet IJ30 is described with reference to the accompanying drawings in which:

Fig. C30.1 illustrates a cross section of a single nozzle and actuator of a 1600 dpi print head, shown during the firing of a drop. When the paddle is actuated, air vents in through holes in a hydrophobic PTFE grill under the paddle. This prevents a low pressure regionforming under the paddle and holding it back.
Fig. C30.2 illustrates an exploded view of the nozzle.
Fig. C30.3 illustrates detail of the doubly serpentine heater.

IJ30 has one thermoelastic bend actuator for each nozzle. One end of the actuator is mechanically connected to the substrate, and the other end is connected to a stiff paddle. When energized, the actuator bends away from the substrate towards a nozzle, causing the attached paddle to eject ink. As the actuator is cooled by the ink, it returns to its nominal position. The nozzle is fabricated as a hole in an integrated nozzle plate of silicon nitride. The nozzle plate is raised from the substrate to form the ink channels, nozzles, filters, and nozzle chambers by the use of a sacrificial oxide.

The actuator is composed of a copper heater embedded in a polytetrafluoroethylene (PTFE) paddle. PTFE is used because it has a very high coefficient of thermal expansion.

The copper heater is formed as a serpentine wire of approximately 0.3 µm thickness. The copper is deposited over thickness corrugations of the PTFE, and then etched in the serpentine pattern. The PTFE corrugations can be formed by exposing a resist layer to a halftone mask, then etching the resist and the underlying PTFE at the same rate. As a result the copper heater 'winds' throughout the volume of the PTFE. This is done to increase the response speed of the actuator, by reducing the thermal distance between the heater and the PTFE of the actuator. After heater deposition and etching, a further 3 µm of PTFE is deposited to make the cold side of the bend actuator. The upper surface of the PTFE is treated to make it hydrophilic.

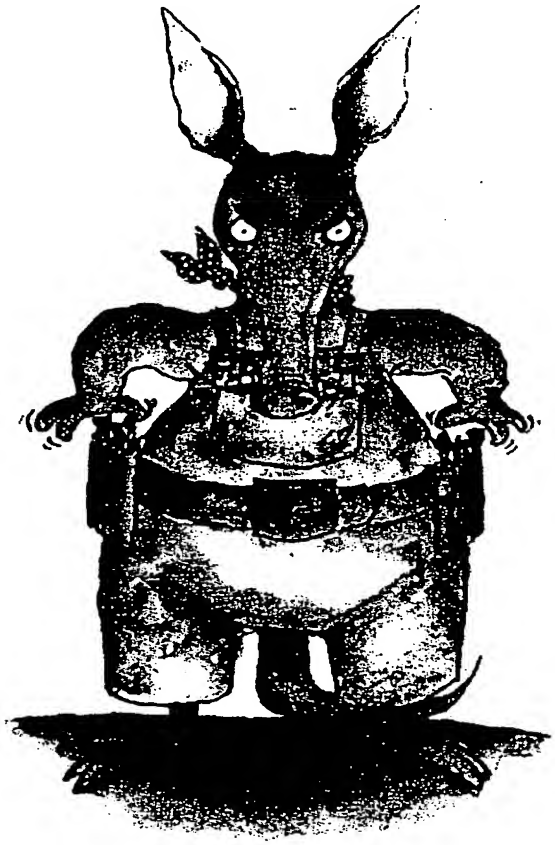
When data signals distributed on the print head indicate that a particular nozzle is to eject a drop of ink, the drive transistor for that nozzle is turned on for around 3 µs. This energizes the heater which

heats the lower half of the PTFE. The top half is not heated, and is cooled by the water based ink. The lower PTFE expands rapidly. This expansion causes the actuator to bend, moving the paddle which pushes ink out of the nozzle. There is an air bubble between the paddle and the substrate, which forms due to the hydrophobic nature of the PTFE on the back surface of the paddle. An air vent connects the air bubble to the ambient air, allowing the paddle to move without being held back by a large pressure difference under the paddle. The air bubble also prevents the water based ink contacting the underside of the paddle, achieving a higher temperature with lower power.

When the heater current is turned off, the paddle begins to return to its quiescent position. The paddle return 'sucks' some of the ink back into the nozzle, causing the ink drop to break off from the ink in the nozzle. The ink drop then continues towards the recording medium. The return of the paddle forces the air under the paddle back out of the vents.

The drop firing rate is around 7 KHz. The ink jet head is suitable for fabrication as a monolithic pagewide print head.

# Vark User Guide v2.0



**Paul Lapstun, July 1997**

Silverbrook Research Pty Ltd
393 Draling Street, Balmain
NSW 2041 Australia
Phone: +61 2 9550 9777
Fax: +61 2 9550 9201
Email: paull@silverbrook.com.au

*1 Overview* 9
1.1 Introduction 9
1.2 Organization of This User Guide 9
1.3 Syntax Notation 9
1.4 The Vark Environment 10
1.5 The Vark Model Space 10
1.6 The Vark Color Space 10
*2 Artistic Tiling and Painting* 11
2.1 Artistic Tiling and Basic Artistic Painting 11
2.2 Advanced Artistic Painting 13
*3 Geometric Data Types* 14
AFFINE_TRANSFORM 14
PATH 14
PATH_NAVIGATOR 15
POINT 16
RECTANGLE 16
RECTANGLE_LIST 17
VECTOR 17
*4 Geometric Functions* 19
+(POINT, VECTOR) 19
+(VECTOR, VECTOR) 19
-(POINT, POINT) 19
-(POINT, VECTOR) 19
-(VECTOR) 19
-(VECTOR, VECTOR) 20
*(AFFINE_TRANSFORM, AFFINE_TRANSFORM) 20
*(PATH, AFFINE_TRANSFORM) 20
*(POINT, AFFINE_TRANSFORM) 20
*(RECTANGLE, AFFINE_TRANSFORM) 20
*(VECTOR, AFFINE_TRANSFORM) 21
*(VECTOR, real) 21
*(VECTOR, VECTOR) 21
/(VECTOR, real) 21
ANGLE(VECTOR) 22
AT_END(PATH_NAVIGATOR) 22
CENTER(RECTANGLE) 22

CLOSE(PATH) 22
CURVE_TO(PATH, POINT, POINT, POINT) 22
CURVE_TO(PATH, VECTOR, VECTOR, VECTOR) 23
INVERT(AFFINE_TRANSFORM) 23
IS_NULL(RECTANGLE) 23
IS_NULL(VECTOR) 23
LENGTH(VECTOR) 24
LINE_TO(PATH, POINT) 24
LINE_TO(PATH, VECTOR) 24
MAX(RECTANGLE) 24
MIN(RECTANGLE) 24
MOVE_TO(PATH, POINT) 25
MOVE_TO(PATH, VECTOR) 25
NORMAL(VECTOR) 25
NORMALIZE(VECTOR) 25
POINT(PATH_NAVIGATOR) 25
ROTATE(AFFINE_TRANSFORM, real) 26
ROTATE(POINT, real) 26
ROTATE(RECTANGLE, real) 26
ROTATE(VECTOR, real) 26
SCALE(AFFINE_TRANSFORM, real, real) 26
SCALE(POINT, real, real) 27
SCALE(RECTANGLE, real, real) 27
SCALE(VECTOR, real, real) 27
SCALE_CENTERED(RECTANGLE, real, real) 27
SHEAR(AFFINE_TRANSFORM, real, real) 28
SHEAR(POINT, real, real) 28
SHEAR(RECTANGLE, real, real) 28
SHEAR(VECTOR, real, real) 28
SIZE(RECTANGLE) 29
TANGENT(PATH_NAVIGATOR) 29
TRANSFORM(PATH, AFFINE_TRANSFORM) 29
TRANSFORM(POINT, AFFINE_TRANSFORM) 29
TRANSFORM(RECTANGLE, AFFINE_TRANSFORM) 29
TRANSFORM(VECTOR, AFFINE_TRANSFORM) 30
TRANSLATE(AFFINE_TRANSFORM, real, real) 30
TRANSLATE(POINT, real, real) 30

TRANSLATE(RECTANGLE, real, real) 30
VECTOR(POINT) 31
X(POINT) 31
X(VECTOR) 31
Y(POINT) 31
Y(VECTOR) 31
*5 Mathematical Functions* *32*
ABSOLUTE(real) 32
ABSOLUTE(integer) 32
ARC_COSINE(real) 32
ARC_SINE(real) 32
ARC_TANGENT(real) 32
COSINE(real) 33
SINE(real) 33
TANGENT(real) 33
*6 Image Processing Data Types* *34*
BRUSH 34
BUMP_MAP 35
CHANNEL 35
CLIP_IMAGE 36
CLUT 37
CLUT_3D 37
COLOR 38
COLOR_LIST 38
COLOR_SPACE 38
DISPLACEMENT_MAP 39
DITHER MATRIX 39
FACE 40
FACE_LIST 40
FILE_NAME 40
FONT 40
FONT_NAME 41
FONT_STYLE 41
ILLUMINATION_OPTION 41
ILLUMINATION_OPTIONS 42
IMAGE 42
IMAGE_FONT 46

IMAGE_FONT_3D 46
KERNEL 47
LIGHT 47
LIGHT_LIST 48
MATTE 48
PAINT 49
PALETTE 50
TEXT 50
TEXT_OPTION 50
TEXT_OPTIONS 51
TILE 51
TILE_LAYER 52
TILE_LAYER_LIST 52
TILE_LIST 52
TILE_OPTION 52
TILE_OPTIONS 54
TILE_PATTERN 54
VECTOR_MAP 55
WARP_MAP 55
*7 Image Processing Functions* *57*
ADD(IMAGE, COLOR) 57
ADD(IMAGE, IMAGE) 57
ADD_NOISE(IMAGE, real) 57
ADD_MONOCHROME_NOISE(IMAGE, real) 57
AREA_THRESHOLD(IMAGE, real) 58
ART_TILE(IMAGE, MATTE, TILE_LAYER_LIST, TILE_OPTIONS) 58
ART_TILE(IMAGE, TILE_PATTERN, TILE_OPTIONS) 58
AXIS_POINT(CLIP_IMAGE) 59
BLUR(IMAGE) 59
BOTTOM_EDGE() 59
CHARACTER_ADVANCE(character, FONT) 59
CHARACTER_ADVANCE(character, IMAGE_FONT) 60
COLOR_BLEND(COLOR, COLOR, COLOR, COLOR, COLOR_SPACE) 60
COMPOSITE(CLIP_IMAGE, IMAGE, POINT, real) 60
COMPOSITE(IMAGE, MATTE, IMAGE) 61
CONVERT_COLOR_SPACE(IMAGE, COLOR_SPACE) 61
CONVOLVE(IMAGE, KERNEL, boolean) 61

CROP(BUMP_MAP) 62
CROP(CLIP_IMAGE) 62
CROP(DISPLACEMENT_MAP) 62
CROP(IMAGE) 62
CROP(IMAGE, real, real) 63
CROP(IMAGE, RECTANGLE) 63
CROP(MATTE) 63
DESATURATE(IMAGE) 63
DETECT_DETAILED_FACE(IMAGE) 64
DETECT_DETAILED_FACES(IMAGE, integer, integer, real) 64
DETECT_FACE(IMAGE) 64
DETECT_FACES(IMAGE, integer, integer, real) 64
DITHER(IMAGE, DITHER_MATRIX, COLOR_LIST) 65
EXTRACT_CHANNEL(IMAGE, CHANNEL) 65
FIND_EDGES(IMAGE) 65
FLIP_HORIZONTALLY(IMAGE) 66
FLIP_VERTICALLY(IMAGE) 66
HORIZONTAL_COLOR_BLEND(COLOR, COLOR, COLOR_SPACE) 66
ILLUMINATE(IMAGE, ILLUMINATION_OPTIONS, LIGHT_LIST, BUMP_MAP) 66
INVERT(IMAGE) 68
LEFT_EDGE() 68
LOOKUP_COLOR(IMAGE, CLUT) 68
LOOKUP_COLOR(IMAGE, CLUT_3D) 68
MAX(IMAGE, IMAGE) 68
MAX_FILTER(IMAGE, integer) 69
MEAN_MIN_MAX_FILTER(IMAGE, integer) 69
MEDIAN_FILTER(IMAGE, integer) 69
MIN(IMAGE, IMAGE) 69
MIN_FILTER(IMAGE, integer) 70
MULTIPLY(IMAGE, COLOR) 70
MULTIPLY(IMAGE, IMAGE) 70
MULTIPLY(MATTE, real) 70
PHOTO() 71
POSTERIZE(IMAGE, integer) 71
PRINT(IMAGE) 71
RESIZE(BUMP_MAP, VECTOR) 71

RESIZE(CLIP_IMAGE, VECTOR) 71
RESIZE(DISPLACEMENT_MAP, VECTOR) 72
RESIZE(IMAGE, VECTOR) 72
RESIZE(MATTE, VECTOR) 72
RIGHT_EDGE() 72
QUANTIZE(IMAGE, PALETTE) 72
QUANTIZE_HUE(IMAGE, integer) 73
RAW_PHOTO() 73
RENDER_TEXT(TEXT, FONT, COLOR, TEXT_OPTIONS) 73
RENDER_TEXT(TEXT, IMAGE_FONT, TEXT_OPTIONS) 73
RENDER_TEXT(TEXT, IMAGE_FONT, TEXT_OPTIONS) 74
REPLACE_CHANNEL(IMAGE, CHANNEL, IMAGE) 74
REPLACE_CHANNEL(IMAGE, CHANNEL, real) 74
ROTATE(CLIP_IMAGE, real) 75
ROTATE(IMAGE, real) 75
SAVE(CLUT_3D, FILE_NAME) 75
SAVE(IMAGE, FILE_NAME) 75
SAVE(KERNEL, FILE_NAME) 75
SAVE(PALETTE, FILE_NAME) 76
SCALE(CLIP_IMAGE, real) 76
SCALE(CLIP_IMAGE, real, real) 76
SCALE(IMAGE, real) 76
SCALE(IMAGE, real, real) 77
SHADOW(CLIP_IMAGE, real, real, COLOR) 77
SIZE(CLIP_IMAGE) 77
SIZE(IMAGE) 77
SKELETONIZE(IMAGE) 78
SKELETONIZE_TO_PATH(IMAGE) 78
STROKE(PATH, BRUSH, COLOR, IMAGE) 78
STROKE(PATH, BRUSH, PAINT, IMAGE) 78
SUBTRACT(IMAGE, COLOR) 79
SUBTRACT(IMAGE, IMAGE) 79
TESSELATE(BUMP_MAP) 79
TESSELATE(IMAGE) 80
THRESHOLD(IMAGE, real) 80
TOP_EDGE() 80
TRANSLATE(IMAGE, real, real) 80

VERTICAL_COLOR_BLEND(COLOR, COLOR, COLOR_SPACE) 80
WARP(CLIP_IMAGE, WARP_MAP, boolean) 81
WARP(IMAGE, WARP_MAP, boolean, RECTANGLE_LIST) 81
X_SIZE() 82
Y_SIZE() 82
*8 Vark Language Reference* *83*
8.1 Syntax Notation 83
8.2 Lexical Structure 83
8.3 Program Structure 84
8.4 Declarations and Constants 84
8.5 Expressions and Statements 87
8.6 Functions and Procedures 91
8.7 Exception Handling 93
*9 Vark Syntax Summary* *94*
*10 File Formats* *98*
10.1 Brush Stroke Segment File 98
10.2 Bump Map File 98
10.3 Clip Image File 98
10.4 Color Lookup Table (CLUT) File 98
10.5 3D Color Lookup Table (3D CLUT) File 98
10.6 Displacement Map File 99
10.7 Dither Matrix File 99
10.8 Font 99
10.9 Image File 99
10.10 Image Font File 99
10.11 Image Font 3D File 99
10.12 Kernel File 99
10.13 Matte File 99
10.14 Palette File 99
10.15 Path File 99
10.16 Tile File 100
10.17 Vector Map File 100
10.18 Warp Map File 100
*11 VarkShop Reference* *101*
*12 VarkShow Reference* *102*

# 1 Overview

## 1.1 INTRODUCTION

Vark is a general-purpose programming language. It provides a range of primitive data types (integer, real, boolean, character); it provides a range of aggregate data types (array, string, record) for constructing more complex types; it provides a rich set of arithmetic and relational operators; it supports conditional and iterative control flow (if-then-else, while-do); and it supports recursive functions and procedures.

Vark is a strongly typed language. Valid program syntax therefore more readily leads to valid program semantics.

Vark comes with a portable interpreter written in C++. It therefore supports programming to multiple platforms, including platforms not yet invented, without re-targeting.

Vark is also a general-purpose image-processing language. It provides a range of image-processing data types (image, clip image, matte, color, color lookup table, palette, dither matrix, convolution kernel, etc.), graphics data types (font, text, path), a set of image-processing functions (color transformations, compositing, filtering, spatial transformations and warping, illumination, text setting and rendering), and a set of higher-level artistic functions (tiling, painting and stroking).

The image-processing capabilities of Vark are implemented as a set of extensions – i.e. additional built-in data types and functions - to the basic Vark language.

Because Vark is designed to map onto a pipelined image processing hardware architecture, it is function-centric rather than object-centric.

A Vark program is portable in two senses. It is independent of the CPU and image processing engines of its host because it is interpreted. And it is independent of the input color characteristics and resolution of the host input device, and the output color characteristics and resolution of the host output device, because it uses a device-independent model space and a device-independent color space.

## 1.2 ORGANIZATION OF THIS USER GUIDE

Section 0 provides an overview of Vark. Section 2 descusses the use of the artistic tiling and painting functions of Vark. Sections 3 through 7 describe high-level Vark data types and functions : geometric data types, geometric functions, mathematical functions, image processing data types, and image processing functions. Sections 8 and 9 describe the Vark language. Section 10 describes Vark data file formats specific to the Vark demonstration system. Section 11 describes VarkShop, the Vark demonstration application for Windows 95/NT.

Section 0 - Overview, Section 6 – Image Processing Data Types, Section 10 – File Formats, and Section 11 – VarkShop Reference, together provide the minimum information required for the confident reader to dive straight into programming in Vark.

Note that data types and the functions which operate on them are define in successive sections, each organised alphabetically. Each data type definition includes a list of functions which operate on the data type; these are then defined in the succeeding section. Constructor functions – overloaded functions which have the same name as the data type and which construct new instances of the data type – are defined in the data type section.

## 1.3 SYNTAX NOTATION

The following sections describe Vark syntax using type declarations and function declarations. All actual Vark syntax is shown in `constant width type`. Function names

are also shown in **bold**, and formal parameter names in *italics*, for clarity. Optional parameters are shown enclosed in square brackets.

## 1.4 THE VARK ENVIRONMENT

A Vark program typically embodies the transformation of an input image into an output image. Vark therefore provides a special function `PHOTO()` which returns the current input image, and a special procedure `PRINT(IMAGE Image)` which prints (or otherwise outputs) the specified image.

The following Vark program, then, simply prints the input image:

```
PRINT(PHOTO());
```

A Vark program may contain meta-variables – variables whose values are intended to be supplied from the environment, perhaps via some interaction with the user. For example, if a Vark program represents a greeting card, then the variable containing the canned message text may be declared as a meta-variable, allowing the Vark interpreter to offer the text to the user for editing before it executes the program.

## 1.5 THE VARK MODEL SPACE

The Vark model space is a continuous and conceptually infinite right-handed three-dimensional rectangular coordinate space. Its scale is the same in all three dimensions.

The photo is mapped to the model space so that it lies in the x-y plane with its center at the origin, with 'right' pointing in the x direction and 'up' pointing in the y direction, facing in the z direction.

The photo is mapped to the model space so that its smaller dimension maps to the coordinate range –1.0 to +1.0, and its larger dimension maps to the coordinate range $-r$ to $+r$, where $r$ is the ratio of the its larger dimension to its smaller dimension. This defines the *working region*. The working region has a size of $2 \times 2r$ or $2r \times 2$, depending on the aspect ratio of the image. This defines the *working size*. The number of pixels in the photo per unit model space defines the *working resolution*.

This fitting scheme allows image effects to be designed to be visible independently of the aspect ratio of the image. More careful fitting of an effect to the image is possible by interrogating the logical size of the photo.

A Vark program may combine the photo with other images and clip images. Each such image has a pixel size and a resolution which together specify its logical size in model space. When the image is loaded it is automatically scaled so that its resolution matches the working resolution. This ensures that its logical size in model space is maintained.

## 1.6 THE VARK COLOR SPACE

Vark uses the CIE L*a*b* color space, defined on a finite and continuous three-dimensional rectangular coordinate space where the axes correspond to the lightness (L*) and chrominance (*a and *b) components. Lightness is defined on the inclusive range 0.0 to 1.0. Chrominance is defined on the inclusive range –1.0 to 1.0. This defines the *working color space*.

CIE L*a*b* is independent of any particular hardware device, and is more perceptually uniform and complete than other widely used color spaces. It provides the ideal intermediate color space in a system which includes image capture, image processing, and image output.

Vark also supports the use of other color spaces, such as RGB and HSV, for particular effects. Each color space is defined in reference to CIE L*a*b*. Each color space is defined in continuous coordinates. Most color components are defined on the inclusive range 0.0 to 1.0. Polar components (such as hue) are defined on the polar range 0.0 to 360.0 degrees, where 0.0 is equivalent to 360.0.

# 2 Artistic Tiling and Painting

## 2.1 Artistic Tiling and Basic Artistic Painting

Artistic tiling converts a color image into a tiled color image. It places translucent textured tiles in a regular arrangement in the output image, deriving the color of each tile from the color of the corresponding pixels in the input image.

Tiles may have any shape, but are typically designed to fit together so that they fully tile an area, i.e. leaving no gaps between adjacent tiles. A set of tile shapes typically fit together to form a larger pattern which itself fully tiles an area when repeated.

Artistic tiling is embodied in the `ART_TILE` functions.

### 2.1.1 The Tiling Process

The input to the tiling process is a color image to be reproduced as a tiling, described by the `IMAGE` data type, and a tile pattern, described below. The output of the tiling process is a tiled rendition of the input image. Additional optional parameters are described in subsequent sections.

A tile pattern is defined by an offset between successive placements of the pattern (the inter-pattern offset), a list of tiles which make up the pattern, and the offsets between successive placements of the tiles (the inter-tile offsets). A tile pattern is described by the `TILE_PATTERN` data type.

A tile is defined by an image with two channels. The first channel defines the tile's shape and opacity, i.e. a non-zero opacity value identifies a pixel which is part of the tile, and the opacity of that pixel. The second channel defines the tile's surface texture as a height field or bump map. A tile is described by the `TILE` data type.

All offsets are measured in tile pixels for convenience. However, like all Vark objects, tile patterns and tiles have a logical size and are resolution-independent.

The tiling process starts in the top left corner of the input image. It proceeds from left to right and from top to bottom, repeatedly placing the tiles which make up the tile pattern, spaced according to the specified inter-tile and inter-pattern offsets.

### 2.1.2 Tile Coloring and Compositing

Each tile has a shape which is defined by its shape/opacity channel. The tile's shape and the tile's placement in the image together determine a set of pixels which are covered by the tile in the input image. The color of the tile in the output image is a function of this set of pixels. Coloring functions include copying the input colors pixel-by-pixel, which is the default, and taking the average of the input colors[1].

The colored tile is composited with the output image according to the tile's opacity. The output image is first initialised to black, or alternatively to some arbitrary image[2], e.g. the input image itself.

### 2.1.3 Tile Surface Texture

Each tile has a surface texture which is defined by its texture channel. The texture channel defines the relative surface height of each pixel in the tile.

This height field is later used to compute surface normals which in turn are used to determine the angles of reflection of simulated incident light. When the tile, or the image in

[1] refer to the `TILE_OPTION` constructor **`TILE_AVERAGE_COLOR`**

[2] refer to the `TILE_OPTION` constructor **`TILE_OVER_BACKGROUND`**

which the tile is placed, is subject to simulated lighting[1], the surface texture thus becomes apparent.

When texture generation is enabled[2], the tile's texture is, by default, written directly to the output image's texture channel, replacing any existing texture.

### 2.1.4 Using the Tiling Process for Basic Painting

The tiling process lends itself to producing basic painting effects. Simply treat a tile pattern as a collection of brush strokes. Design the tile images to look like brush strokes, e.g. with streaks from the brush hairs, and greater thickness at the end of the stroke, etc. Set the inter-tile and inter-pattern offsets so that the strokes overlap. Specify that the stroke collection (i.e. tile pattern) is random[3], so that the tiling process selects strokes from the stroke collection randomly, rather than iterating through the tiles in the pattern sequentially. Use one or more of the texturing, positioning and coloring effects described below.

### 2.1.5 Multi-Layer Painting

There are typically various levels of detail in an input image which require different levels of detail when reproduced via the tiling/painting process. This may be achieved by processing the image in multiple passes, using a different scale stroke collection in each pass, and compositing the results of each pass with the (cumulative) background. An image-sized detail map, consisting of a per-pixel detail measure, may be used to control which parts of the image each pass processes. Each pass is given a detail threshold above which it processes the image. Successive passes are given increasing levels of detail, and decreasing stroke collection scales. The first pass thus processes the entire image, the second pass adds a certain level of detail, the third pass adds still more detail, and so on. This is somewhat analogous to how a human painter refines a painting.

The multi-layer version of the **ART_TILE** function accepts a list of tile layer descriptions, each consisting of a tile pattern and a detail threshold, and a detail or layer selection map. A tile layer is described by the TILE_LAYER data type.

A detail map may be automatically generated from the input image by locating edges in the image[4].

### 2.1.6 Stroke Texturing Effects

A tile is conceptually rigid. Its surface texture does not interact with the surface texture of the background or of any tiles it might overlap (though it would typically not overlap any other tiles). A brush stroke, on the other hand, is conceptually plastic. Its thickness and surface texture are affected by the surface texture of the background and of any other brush strokes it might overlap. Specifically, the brush stroke tends to fill in depressions in the background, but tends itself to become thinner where it lands on peaks in the background. (The background is taken to include the effect of any brush strokes which precede the stroke in question).

When a brush stroke is laid down, therefore, its texture is combined with the texture of the background in one of several ways. (1) The stroke height is added to a proportion (25%) of the background height to yield the new background height[5], but the height is constrained not to diminish. (2) The average height of the background under the stroke is computed[6]. At a particular pixel, if the background height is less than the average, then the stroke height is simply added to the background height. Thus the stroke fills in back-

---

[1] refer to the **ILLUMINATE** function

[2] refer to the TILE_OPTION constructor **TILE_GENERATE_BUMP**

[3] refer to the TILE_PATTERN constructor **RANDOM_TILE_PATTERN**

[4] refer to the **FIND_EDGES** function

[5] refer to the TILE_OPTION constructor **TILE_SUM_BUMPS**

[6] refer to the TILE_OPTION constructor **TILE_AVERAGE_BUMP**

ground depressions. If the background height is greater than (or equal to) the average, then the stroke height is added to the average. Thus the stroke is thinned by background peaks. To prevent the background from actually breaking through the stroke (i.e. thinning it to zero), the height is constrained to increase by a minimum amount - the minimum stroke thickness (which may of course be zero).

The opacity of the stroke may be scaled (on a pixel-by-pixel basis) to account for the net thinning of the stroke which results from the texture combining algorithm. Thus where the stroke is thinned by peaks in the background it also becomes more transparent.

### 2.1.7 Tile/Stroke Positioning Effects

Once the tile/stroke position is determined, it may be manipulated further for artistic effect. (1) The position may be randomly jittered within a specified range at a specified rate to simulate natural variations in manual tile or stroke positioning[1]. The tile jitter range is typically limited to prevent tiles from overlapping, - i.e. only the width of the inter-tile grouted gap varies. The stroke jitter range is typically less limited since brush strokes typically do overlap. (2) The position may be displaced according to a displacement map to simulate more regular variations in tile or stroke position[2]. The displacement map consists of two channels. The first contains column displacements. The second contains row displacements.

### 2.1.8 Tile/Stroke Coloring Effects

Once the tile/stroke color is determined, it may be manipulated further for artistic effect. (1) The color may be mapped to an arbitrary palette to simulate a limited set of tiles or a limited paint palette[3]. If the color is mapped to a palette, then the overall image reproduction may be improved by diffusing the color error locally into the remaining part of the input image[4]. (2) The color may be randomly jittered within a specified range at a specified rate to simulate tile color imperfections or paint mixing variations[5]. (3) The chrominance component of the color may be randomly inverted at a specified rate to simulate impressionism's use of color opposites[6].

## 2.2 ADVANCED ARTISTIC PAINTING

[TBA]

---

[1] refer to the `TILE_OPTION` constructor **`TILE_JITTER_POSITION`**

[2] refer to the `TILE_OPTION` constructor **`TILE_DISPLACE_POSITION`**

[3] refer to the `TILE_OPTION` constructor **`TILE_MAP_COLOR_TO_PALETTE`**

[4] refer to the `TILE_OPTION` constructor **`TILE_DIFFUSE_COLOR_ERROR`**

[5] refer to the `TILE_OPTION` constructor **`TILE_JITTER_COLOR`**

[6] refer to the `TILE_OPTION` constructor **`TILE_INVERT_CHROMA`**

# 3 Geometric Data Types

## AFFINE_TRANSFORM

### TYPE NAME

AFFINE_TRANSFORM

### CONSTRUCTOR FUNCTIONS

AFFINE_TRANSFORM **AFFINE_TRANSFORM**()

### DESCRIPTION

An AFFINE_TRANSFORM specifies a arbitrary 2D affine transform.

An AFFINE_TRANSFORM consists of the identity transform when first constructed. It is built up by multiplying it by one or more primitive scale, shear, rotate or translate transforms, or by other arbitrary affine transforms.

Affine transform related functions are listed below. They are described in the next section.

### TRANSFORMATION OPERATORS AND FUNCTIONS

```
AFFINE_TRANSFORM *(
     AFFINE_TRANSFORM Transform1,
     AFFINE_TRANSFORM Transform2)
AFFINE_TRANSFORM INVERT(AFFINE_TRANSFORM Transform)
AFFINE_TRANSFORM SCALE(
     AFFINE_TRANSFORM Transform,
     real X_Scale,
     real Y_Scale)
AFFINE_TRANSFORM SHEAR(
     AFFINE_TRANSFORM Transform,
     real X_Shear,
     real Y_Shear)
AFFINE_TRANSFORM ROTATE(
     AFFINE_TRANSFORM Transform,
     real Angle)
AFFINE_TRANSFORM TRANSLATE(
     AFFINE_TRANSFORM Transform,
     real X_Offset,
     real Y_Offset)
```

## PATH

### TYPE NAME

PATH

### CONSTRUCTOR FUNCTIONS

```
PATH PATH()
PATH PATH(FILE_NAME File_Name)
```

### PARAMETERS

| | |
|---|---|
| *File_Name* | Name of file containing path<br>(refer to Section 10.15 for format of file) |

### DESCRIPTION

A PATH specifies a set of zero or more sub-paths, each of which specifies a continuous set of points in the x-y plane.

Each sub-path of a path is constructed from line and curve segments joined end-to-end. Each curve segment is a cubic Bézier curve.

A path maintains a current point, relative to which line and curve segments are added. The current point is the end-point of the most recent LINE_TO or CURVE_TO procedure call, or the point of the most recent MOVE_TO procedure call. It is initialised to the origin.

A sub-path is either open or closed. If it is closed, its last segment is considered continuous with its first segment. When the sub-path is stroked with a brush, for example, this means that end-caps are not applied. A sub-path is not considered closed just because its end-point is coincident with its start-point. It must be explicitly closed by calling the CLOSE procedure.

Path related functions are listed below. They are described in the next section.

#### PATH CONSTRUCTION PROCEDURES

```
MOVE_TO(variable PATH Path, VECTOR Offset)
MOVE_TO(variable PATH Path, POINT Point)
LINE_TO(variable PATH Path, VECTOR Offset)
LINE_TO(variable PATH Path, POINT Point)
CURVE_TO(
     variable PATH Path,
     VECTOR Offset1,
     VECTOR Offset2,
     VECTOR Offset3)
CURVE_TO(
     variable PATH Path,
     POINT Point1,
     POINT Point2,
     POINT Point3)
CLOSE(variable PATH Path)
```

#### TRANSFORMATION FUNCTIONS AND PROCEDURES

```
PATH TRANSFORM(PATH Path, AFFINE_TRANSFORM Transform)
PATH *(PATH Path, AFFINE_TRANSFORM Transform)
TRANSFORM(variable PATH Path, AFFINE_TRANSFORM Transform)
```

## PATH_NAVIGATOR

#### TYPE NAME

PATH_NAVIGATOR

#### CONSTRUCTOR FUNCTIONS

```
PATH_NAVIGATOR PATH_NAVIGATOR(PATH Path)
```

#### PARAMETERS

| | |
|---|---|
| *Path* | Path to navigate |

#### DESCRIPTION

A PATH_NAVIGATOR provides a means of stepping along a path.

A PATH_NAVIGATOR maintains a current position along the path being navigated, returns information about the current position, such as the actual point and the tangent at that point, and allows the position to be advanced by some distance, or reset to the start of the path.

Path navigator related functions are listed below. They are described in the next section.

#### QUERY FUNCTIONS

```
boolean AT_END(PATH_NAVIGATOR Path_Navigator)
POINT POINT(PATH_NAVIGATOR Path_Navigator)
VECTOR TANGENT(PATH_NAVIGATOR Path_Navigator)
```

### NAVIGATION PROCEDURES

```
START(variable PATH_NAVIGATOR Path_Navigator)
SKIP(variable PATH_NAVIGATOR Path_Navigator)
ADVANCE(
     variable PATH_NAVIGATOR Path_Navigator,
     real Distance)
```

## POINT

### TYPE NAME

POINT

### CONSTRUCTOR FUNCTIONS

```
POINT POINT(real X, real Y)
POINT POINT(VECTOR Vector)
POINT ORIGIN()
```

### PARAMETERS

| | |
|---|---|
| *X* | Coordinate in x dimension |
| *Y* | Coordinate in y dimension |
| *Vector* | Origin vector |

### DESCRIPTION

A POINT specifies an absolute position in the x-y plane.

A POINT is constructed from an x and y coordinate pair, or from a vector. Both the x and y pair and the vector describe the offset of the point from the origin.

ORIGIN constructs a point at the origin.

Point related functions are listed below. They are described in the next section.

### ATTRIBUTE FUNCTIONS

```
real X(POINT Point)
real Y(POINT Point)
VECTOR VECTOR(POINT Point)
```

### TRANSFORMATION OPERATORS AND FUNCTIONS

```
POINT +(POINT Point, VECTOR Vector)
POINT -(POINT Point, VECTOR Vector)
VECTOR -(POINT Point1, POINT Point2)
POINT SCALE(POINT Point, real X_Scale, real Y_Scale)
POINT SHEAR(POINT Point, real X_Shear, real Y_Shear)
POINT ROTATE(POINT Point, real Angle)
POINT TRANSLATE(POINT Point, real X_Offset, real Y_Offset)
POINT *(POINT Point, AFFINE_TRANSFORM Transform)
```

## RECTANGLE

### TYPE NAME

RECTANGLE

### CONSTRUCTOR FUNCTIONS

```
RECTANGLE RECTANGLE(
     real Min_X,
     real Min_Y,
     real Max_X,
     real Max_Y)
RECTANGLE RECTANGLE(POINT Point1, POINT Point2)
```

### PARAMETERS

| | |
|---|---|
| *Min_X* | X coordinate of minimum point |
| *Min_Y* | Y coordinate of minimum point |
| *Max_X* | X coordinate of maximum point |

| | |
|---|---|
| *Max_Y* | Y coordinate of maximum point |
| *Point1* | First corner point |
| *Point2* | Second corner point |

**DESCRIPTION**

A RECTANGLE describes an absolute rectangular region in the x-y plane.

A RECTANGLE is constructed from minimum and a maximum x and y coordinates, or from a pair of corner points. In the former case, if the minimum x coordinate exceeds the maximum x coordinate, or the minimum y coordinate exceeds the maximum y coordinate, the the rectangle is defined as null – i.e. defining no region. In the latter case, the two points define corners of the rectangle, and so always define a non-null rectangle, though it may be zero-sized.

Rectangle related functions are listed below. They are described in the next section.

**ATTRIBUTE FUNCTIONS**

```
POINT MIN(RECTANGLE Rect)
POINT MAX(RECTANGLE Rect)
POINT CENTER(RECTANGLE Rect)
VECTOR SIZE(RECTANGLE Rect)
boolean IS_NULL(RECTANGLE Rect)
```

**TRANSFORMATION FUNCTIONS**

```
RECTANGLE SCALE(RECTANGLE Rect, real X_Scale, real Y_Scale)
RECTANGLE SCALE_CENTERED(
     RECTANGLE Rect,
     real X_Scale,
     real Y_Scale)
RECTANGLE SHEAR(RECTANGLE Rect, real X_Shear, real Y_Shear)
RECTANGLE ROTATE(RECTANGLE Rect, real Angle)
RECTANGLE TRANSLATE(
     RECTANGLE Rect,
     real X_Offset,
     real Y_Offset)
RECTANGLE *(RECTANGLE Rect, AFFINE_TRANSFORM Transform)
```

## RECTANGLE_LIST

**TYPE NAME**

RECTANGLE_LIST

**DESCRIPTION**

An alias for `string of` RECTANGLE.

## VECTOR

**TYPE NAME**

VECTOR

**CONSTRUCTOR FUNCTIONS**

```
VECTOR VECTOR(real X, real Y)
VECTOR VECTOR(POINT Point)
```

**PARAMETERS**

| | |
|---|---|
| *X* | Size in x dimension |
| *Y* | Size in y dimension |
| *Point* | Source point |

**DESCRIPTION**

A VECTOR specifies a vector, i.e. a distance and direction, in the x-y plane.

A VECTOR is constructed from an'x and y offset pair, or is extracted from a point.

Vector related functions are listed below. They are described in the next section.

**ATTRIBUTE FUNCTIONS**

```
real X(VECTOR Vector)
real Y(VECTOR Vector)
boolean IS_NULL(VECTOR Vector)
real LENGTH(VECTOR Vector)
real ANGLE(VECTOR Vector)
```

**TRANSFORMATION OPERATORS AND FUNCTIONS**

```
VECTOR -(VECTOR Vector)
VECTOR +(VECTOR Vector1, VECTOR Vector2)
VECTOR -(VECTOR Vector1, VECTOR Vector2)
real *(VECTOR Vector1, VECTOR Vector2)
VECTOR *(VECTOR Vector, real Scale)
VECTOR /(VECTOR Vector, real Scale)
VECTOR NORMAL(VECTOR Vector)
VECTOR NORMALIZE(VECTOR Vector)
VECTOR SCALE(VECTOR Vector, real X_Scale, real Y_Scale)
VECTOR SHEAR(VECTOR Vector, real X_Shear, real Y_Shear)
VECTOR ROTATE(VECTOR Vector, real Angle)
VECTOR *(VECTOR Vector, AFFINE_TRANSFORM Transform)
```

# 4 Geometric Functions

## +(POINT, VECTOR)

**SYNOPSIS**

```
POINT +(POINT Point, VECTOR Vector)
```

**PARAMETERS**

| | |
|---|---|
| *Point* | Source point |
| *Vector* | Offset vector |

**DESCRIPTION**

Returns a copy of the source point offset by the specified vector.

## +(VECTOR, VECTOR)

**SYNOPSIS**

```
VECTOR +(VECTOR Vector1, VECTOR Vector2)
```

**PARAMETERS**

| | |
|---|---|
| *Vector1* | First source vector |
| *Vector2* | Second source vector |

**DESCRIPTION**

Returns the sum of the source vectors.

## -(POINT, POINT)

**SYNOPSIS**

```
VECTOR -(POINT Point1, POINT Point2)
```

**PARAMETERS**

| | |
|---|---|
| *Point1* | First source point |
| *Point2* | Second source point |

**DESCRIPTION**

Returns the vector joining the first point to the second.

## -(POINT, VECTOR)

**SYNOPSIS**

```
POINT -(POINT Point, VECTOR Vector)
```

**PARAMETERS**

| | |
|---|---|
| *Point* | Source point |
| *Vector* | Offset vector |

**DESCRIPTION**

Returns a copy of the source point offset by the negation of the specified vector.

## -(VECTOR)

**SYNOPSIS**

```
VECTOR -(VECTOR Vector)
```

**PARAMETERS**

| | |
|---|---|
| *Vector* | Source vector |

**DESCRIPTION**

Returns the negation of the source vector.

## -(VECTOR, VECTOR)

**SYNOPSIS**

```
VECTOR -(VECTOR Vector1, VECTOR Vector2)
```

**PARAMETERS**

| | |
|---|---|
| *Vector1* | First source vector |
| *Vector2* | Second source vector |

**DESCRIPTION**

Returns the sum of the first source vector and the negation of the second.

## *(AFFINE_TRANSFORM, AFFINE_TRANSFORM)

**SYNOPSIS**

```
AFFINE_TRANSFORM *(
     AFFINE_TRANSFORM Transform1,
     AFFINE_TRANSFORM Transform2)
```

**PARAMETERS**

| | |
|---|---|
| *Transform1* | First transform |
| *Transform2* | Second transform |

**DESCRIPTION**

Returns a copy of the first transform multiplied by the second.

## *(PATH, AFFINE_TRANSFORM)

**SYNOPSIS**

```
PATH *(PATH Path, AFFINE_TRANSFORM Transform)
```

**PARAMETERS**

| | |
|---|---|
| *Path* | Source path |
| *Transform* | Transform |

**DESCRIPTION**

Returns a copy of the source path transformed according to the specified transform.

## *(POINT, AFFINE_TRANSFORM)

**SYNOPSIS**

```
POINT *(POINT Point, AFFINE_TRANSFORM Transform)
```

**PARAMETERS**

| | |
|---|---|
| *Point* | Source point |
| *Transform* | Transform |

**DESCRIPTION**

Returns a copy of the source point transformed according to the specified transform.

## *(RECTANGLE, AFFINE_TRANSFORM)

**SYNOPSIS**

```
RECTANGLE *(RECTANGLE Rect, AFFINE_TRANSFORM Transform)
```

**PARAMETERS**

| | |
|---|---|
| *Rect* | Source rectangle |
| *Transform* | Transform |

**DESCRIPTION**

Returns a copy of the source rectangle transformed according to the specified transform.

## *(VECTOR, AFFINE_TRANSFORM)

**SYNOPSIS**

```
VECTOR *(VECTOR Vector, AFFINE_TRANSFORM Transform)
```

**PARAMETERS**

| | |
|---|---|
| *Vector* | Source vector |
| *Transform* | Transform |

**DESCRIPTION**

Returns a copy of the source vector transformed according to the specified transform.

Note that any translation component of the transform is ignored.

## *(VECTOR, real)

**SYNOPSIS**

```
VECTOR *(VECTOR Vector, real Scale)
```

**PARAMETERS**

| | |
|---|---|
| *Vector* | Source vector |
| *Scale* | Scale factor |

**DESCRIPTION**

Returns a copy of the source vector with its length multiplied by the scale factor.

## *(VECTOR, VECTOR)

**SYNOPSIS**

```
VECTOR *(VECTOR Vector1, VECTOR Vector2)
```

**PARAMETERS**

| | |
|---|---|
| *Vector1* | First source vector |
| *Vector2* | Second source vector |

**DESCRIPTION**

Returns the dot product of the source vectors.

## /(VECTOR, real)

**SYNOPSIS**

```
VECTOR /(VECTOR Vector, real Scale)
```

**PARAMETERS**

| | |
|---|---|
| *Vector* | Source vector |
| *Scale* | Scale factor |

**DESCRIPTION**

Returns a copy of the source vector with its length divided by the scale factor.

## ANGLE (VECTOR)

**SYNOPSIS**

```
real ANGLE(VECTOR Vector)
```

**PARAMETERS**

| | |
|---|---|
| *Vector* | Source vector |

**DESCRIPTION**

Returns the counter-clockwise angle, in degrees, between the source vector and the positive x-axis.

## AT_END (PATH_NAVIGATOR)

**SYNOPSIS**

```
boolean AT_END(PATH_NAVIGATOR Path_Navigator)
```

**PARAMETERS**

| | |
|---|---|
| *Path_Navigator* | Source path navigator |

**DESCRIPTION**

Returns whether the navigator has reached the end of the navigated path.

## CENTER (RECTANGLE)

**SYNOPSIS**

```
POINT CENTER(RECTANGLE Rect)
```

**PARAMETERS**

| | |
|---|---|
| *Rect* | Source rectangle |

**DESCRIPTION**

Returns the center point of the source rectangle.

## CLOSE (PATH)

**SYNOPSIS**

```
CLOSE(variable PATH Path)
```

**PARAMETERS**

| | |
|---|---|
| *Path* | Path to modify |

**DESCRIPTION**

Terminates the current sub-path if non-empty, leaving it closed.

## CURVE_TO (PATH, POINT, POINT, POINT)

**SYNOPSIS**

```
LINE_TO(
    variable PATH Path,
    POINT Point1,
    POINT Point2,
    POINT Point3)
```

**PARAMETERS**

| | |
|---|---|
| *Path* | Path to modify |
| *Point1* | First Bézier control point |
| *Point2* | Second Bézier control point |
| *Point3* | Bézier end-point |

**DESCRIPTION**

Appends a Bézier curve segment to the current sub-path, joining the current point with the specified end-point, with the specified control points. Sets the current point to the curve end-point.

## CURVE_TO(PATH, VECTOR, VECTOR, VECTOR)

**SYNOPSIS**

```
LINE_TO(
    variable PATH Path,
    VECTOR Offset1,
    VECTOR Offset2,
    VECTOR Offset3)
```

**PARAMETERS**

| | |
|---|---|
| *Path* | Path to modify |
| *Offset1* | Offset to first Bézier control point |
| *Offset2* | Offset te second Bézier control point |
| *Offset3* | Offset to Bézier end-point |

**DESCRIPTION**

Adds the specified offsets to the current point to yield the Bézier curve control points and end-point. Appends a Bézier curve segment to the current sub-path, joining the current point with the specified end-point, with the specified control points. Sets the current point to the curve end-point.

## INVERT(AFFINE_TRANSFORM)

**SYNOPSIS**

```
AFFINE_TRANSFORM INVERT(AFFINE_TRANSFORM Transform)
```

**PARAMETERS**

| | |
|---|---|
| *Transform* | Source transform |

**DESCRIPTION**

Returns a copy of the source transform, inverted.

The source transform multiplied by its inverse yields the identity transform.

## IS_NULL(RECTANGLE)

**SYNOPSIS**

```
boolean IS_NULL(RECTANGLE Rect)
```

**PARAMETERS**

| | |
|---|---|
| *Rectangle* | Source rectangle |

**DESCRIPTION**

Returns whether the source rectangle is null.

## IS_NULL(VECTOR)

**SYNOPSIS**

```
boolean IS_NULL(VECTOR Vector)
```

**PARAMETERS**

| | |
|---|---|
| *Vector* | Source vector |

**DESCRIPTION**

Returns whether the source vector is null, i.e. zero-length.

### LENGTH(VECTOR)

**SYNOPSIS**

```
real LENGTH(VECTOR Vector)
```

**PARAMETERS**

| | |
|---|---|
| *Vector* | Source vector |

**DESCRIPTION**

Returns the length of the source vector.

### LINE_TO(PATH, POINT)

**SYNOPSIS**

```
LINE_TO(variable PATH Path, POINT Point)
```

**PARAMETERS**

| | |
|---|---|
| *Path* | Path to modify |
| *Point* | Line end-point |

**DESCRIPTION**

Appends a line segment to the current sub-path, joining the current point with the specified end-point. Sets the current point to the line end-point.

### LINE_TO(PATH, VECTOR)

**SYNOPSIS**

```
LINE_TO(variable PATH Path, VECTOR Offset)
```

**PARAMETERS**

| | |
|---|---|
| *Path* | Path to modify |
| *Offset* | Offset to line end-point |

**DESCRIPTION**

Adds the specified offset to the current point to yield the line end-point. Appends a line segment to the current sub-path, joining the current point with the specified end-point. Sets the current point to the line end-point.

### MAX(RECTANGLE)

**SYNOPSIS**

```
POINT MAX(RECTANGLE Rect)
```

**PARAMETERS**

| | |
|---|---|
| *Rect* | Source rectangle |

**DESCRIPTION**

Returns the maximum, i.e. top right, corner point of the source rectangle.

### MIN(RECTANGLE)

**SYNOPSIS**

```
POINT MIN(RECTANGLE Rect)
```

**PARAMETERS**

| | |
|---|---|
| *Rect* | Source rectangle |

**DESCRIPTION**

Returns the minimum, i.e. bottom left, corner point of the source rectangle.

## MOVE_TO(PATH, POINT)

**SYNOPSIS**

```
MOVE_TO(variable PATH Path, POINT Point)
```

**PARAMETERS**

| | |
|---|---|
| *Path* | Path to modify |
| *Point* | Target point |

**DESCRIPTION**

Terminates the current sub-path if non-empty, leaving it open. Sets the current point to the specified target point.

## MOVE_TO(PATH, VECTOR)

**SYNOPSIS**

```
MOVE_TO(variable PATH Path, VECTOR Offset)
```

**PARAMETERS**

| | |
|---|---|
| *Path* | Path to modify |
| *Offset* | Offset to target point |

**DESCRIPTION**

Adds the specified offset to the current point to yield the target point. Terminates the current sub-path if non-empty, leaving it open. Sets the current point to the specified target point.

## NORMAL(VECTOR)

**SYNOPSIS**

```
VECTOR NORMAL(VECTOR Vector)
```

**PARAMETERS**

| | |
|---|---|
| *Vector* | Source vector |

**DESCRIPTION**

Returns a counter-clockwise normal vector to the source vector.

## NORMALIZE(VECTOR)

**SYNOPSIS**

```
VECTOR NORMALIZE(VECTOR Vector)
```

**PARAMETERS**

| | |
|---|---|
| *Vector* | Source vector |

**DESCRIPTION**

Returns a copy of the source vector with its length normalized to 1.0.

## POINT(PATH_NAVIGATOR)

**SYNOPSIS**

```
POINT POINT(PATH_NAVIGATOR Path_Navigator)
```

**PARAMETERS**

| | |
|---|---|
| *Path_Navigator* | Source path navigator |

**DESCRIPTION**

Returns the navigator's current point on the navigated path.

### ROTATE(AFFINE_TRANSFORM, real)

**SYNOPSIS**

```
AFFINE_TRANSFORM ROTATE(
    AFFINE_TRANSFORM Transform,
    real Angle)
```

**PARAMETERS**

| | |
|---|---|
| *Transform* | Source transform |
| *Angle* | Rotation angle, in degrees |

**DESCRIPTION**

Returns a copy of the source transform multiplied by a transform specifying counter-clockwise rotation about the origin by the specified angle.

### ROTATE(POINT, real)

**SYNOPSIS**

```
POINT ROTATE(POINT Point, real Angle)
```

**PARAMETERS**

| | |
|---|---|
| *Point* | Source point |
| *Angle* | Rotation angle, in degrees |

**DESCRIPTION**

Returns a copy of the source point rotated counter-clockwise about the origin by the specified angle.

### ROTATE(RECTANGLE, real)

**SYNOPSIS**

```
RECTANGLE ROTATE(RECTANGLE Rect, real Angle)
```

**PARAMETERS**

| | |
|---|---|
| *Rect* | Source rectangle |
| *Angle* | Rotation angle, in degrees |

**DESCRIPTION**

Returns a copy of the source rectangle rotated counter-clockwise about the origin by the specified angle.

### ROTATE(VECTOR, real)

**SYNOPSIS**

```
VECTOR ROTATE(VECTOR Vector, real Angle)
```

**PARAMETERS**

| | |
|---|---|
| *Vector* | Source vector |
| *Angle* | Rotation angle, in degrees |

**DESCRIPTION**

Returns a copy of the source vector rotated counter-clockwise by the specified angle.

### SCALE(AFFINE_TRANSFORM, real, real)

**SYNOPSIS**

```
AFFINE_TRANSFORM SCALE(
    AFFINE_TRANSFORM Transform,
    real X_Scale,
    real Y_Scale)
```

**PARAMETERS**

| | |
|---|---|
| *Transform* | Source transform |
| *X_Scale* | Scale factor in x dimension |
| *Y_Scale* | Scale factor in y dimension |

**DESCRIPTION**

Returns a copy of the source transform multiplied by a transform specifying scaling by the specified factors.

## SCALE(POINT, real, real)

**SYNOPSIS**

```
POINT SCALE(POINT Point, real X_Scale, real Y_Scale)
```

**PARAMETERS**

| | |
|---|---|
| *Point* | Source point |
| *X_Scale* | Scale factor in x dimension |
| *Y_Scale* | Scale factor in y dimension |

**DESCRIPTION**

Returns a copy of the source point scaled about the origin by the specified factors.

## SCALE(RECTANGLE, real, real)

**SYNOPSIS**

```
RECTANGLE SCALE(RECTANGLE Rect, real X_Scale, real Y_Scale)
```

**PARAMETERS**

| | |
|---|---|
| *Rect* | Source rectangle |
| *X_Scale* | Scale factor in x dimension |
| *Y_Scale* | Scale factor in y dimension |

**DESCRIPTION**

Returns a copy of the source rectangle scaled about the origin by the specified factors.

## SCALE(VECTOR, real, real)

**SYNOPSIS**

```
VECTOR SCALE(VECTOR Vector, real X_Scale, real Y_Scale)
```

**PARAMETERS**

| | |
|---|---|
| *Vector* | Source vector |
| *X_Scale* | Scale factor in x dimension |
| *Y_Scale* | Scale factor in y dimension |

**DESCRIPTION**

Returns a copy of the source vector scaled by the specified factors.

## SCALE_CENTERED(RECTANGLE, real, real)

**SYNOPSIS**

```
RECTANGLE SCALE_CENTERED(
    RECTANGLE Rect,
    real X_Scale,
    real Y_Scale)
```

**PARAMETERS**

| | |
|---|---|
| *Rect* | Source rectangle |
| *X_Scale* | Scale factor in x dimension |
| *Y_Scale* | Scale factor in y dimension |

**DESCRIPTION**

Returns a copy of the source rectangle scaled about the center of the rectangle by the specified factors.

## SHEAR(AFFINE_TRANSFORM, real, real)

**SYNOPSIS**

```
AFFINE_TRANSFORM SHEAR(
    AFFINE_TRANSFORM Transform,
    real X_Shear,
    real Y_Shear)
```

**PARAMETERS**

| | |
|---|---|
| *Transform* | Source transform |
| *X_Shear* | Shear factor in x dimension |
| *Y_Shear* | Shear factor in y dimension |

**DESCRIPTION**

Returns a copy of the source transform multiplied by a transform specifying shearing by the specified factors.

## SHEAR(POINT, real, real)

**SYNOPSIS**

```
POINT SHEAR(POINT Point, real X_Shear, real Y_Shear)
```

**PARAMETERS**

| | |
|---|---|
| *Point* | Source point |
| *X_Shear* | Shear factor in x dimension |
| *Y_Shear* | Shear factor in y dimension |

**DESCRIPTION**

Returns a copy of the source point sheared by the specified factors.

## SHEAR(RECTANGLE, real, real)

**SYNOPSIS**

```
RECTANGLE SHEAR(RECTANGLE Rect, real X_Shear, real Y_Shear)
```

**PARAMETERS**

| | |
|---|---|
| *Rect* | Source rectangle |
| *X_Shear* | Shear factor in x dimension |
| *Y_Shear* | Shear factor in y dimension |

**DESCRIPTION**

Returns a copy of the source rectangle sheared by the specified factors.

## SHEAR(VECTOR, real, real)

**SYNOPSIS**

```
VECTOR SHEAR(VECTOR Vector, real X_Shear, real Y_Shear)
```

**PARAMETERS**

| | |
|---|---|
| *Vector* | Source vector |
| *X_Shear* | Shear factor in x dimension |
| *Y_Shear* | Shear factor in y dimension |

**DESCRIPTION**

Returns a copy of the source vector sheared by the specified factors.

## SIZE(RECTANGLE)

**SYNOPSIS**

```
VECTOR SIZE(RECTANGLE Rect)
```

**PARAMETERS**

| | |
|---|---|
| *Rectangle* | Source rectangle |

**DESCRIPTION**

Returns the size of the source rectangle, as a vector.

## TANGENT(PATH_NAVIGATOR)

**SYNOPSIS**

```
VECTOR TANGENT(PATH_NAVIGATOR Path_Navigator)
```

**PARAMETERS**

| | |
|---|---|
| *Path_Navigator* | Source path navigator |

**DESCRIPTION**

Returns the tangent vector at the navigator's current point on the navigated path.

## TRANSFORM(PATH, AFFINE_TRANSFORM)

**SYNOPSIS**

```
PATH TRANSFORM(PATH Path, AFFINE_TRANSFORM Transform)
TRANSFORM(variable PATH Path, AFFINE_TRANSFORM Transform)
```

**PARAMETERS**

| | |
|---|---|
| *Path* | Source path |
| *Transform* | Transform |

**DESCRIPTION**

Returns a copy of the source path transformed according to the specified transform. Modifies the path in the procedure version.

## TRANSFORM(POINT, AFFINE_TRANSFORM)

**SYNOPSIS**

```
POINT TRANSFORM(POINT Point, AFFINE_TRANSFORM Transform)
```

**PARAMETERS**

| | |
|---|---|
| *Point* | Source point |
| *Transform* | Transform |

**DESCRIPTION**

Returns a copy of the source point transformed according to the specified transform.

## TRANSFORM(RECTANGLE, AFFINE_TRANSFORM)

**SYNOPSIS**

```
RECTANGLE TRANSFORM(
    RECTANGLE Rect,
    AFFINE_TRANSFORM Transform)
```

**PARAMETERS**

| | |
|---|---|
| *Rect* | Source rectangle |
| *Transform* | Transform |

**DESCRIPTION**

Returns a copy of the source rectangle transformed according to the specified transform.

### TRANSFORM(VECTOR, AFFINE_TRANSFORM)

**SYNOPSIS**

```
VECTOR TRANSFORM(VECTOR Vector, AFFINE_TRANSFORM Transform)
```

**PARAMETERS**

| | |
|---|---|
| *Vector* | Source vector |
| *Transform* | Transform |

**DESCRIPTION**

Returns a copy of the source vector transformed according to the specified transform.

Note that any translation component of the transform is ignored.

### TRANSLATE(AFFINE_TRANSFORM, real, real)

**SYNOPSIS**

```
AFFINE_TRANSFORM TRANSLATE(
      AFFINE_TRANSFORM Transform,
      real X_Offset,
      real Y_Offset)
```

**PARAMETERS**

| | |
|---|---|
| *Transform* | Source transform |
| *X_Offset* | Offset in x dimension |
| *Y_Offset* | Offset in y dimension |

**DESCRIPTION**

Returns a copy of the source transform multiplied by a transform specifying translation by the specified offsets.

### TRANSLATE(POINT, real, real)

**SYNOPSIS**

```
POINT TRANSLATE(POINT Point, real X_Offset, real Y_Offset)
```

**PARAMETERS**

| | |
|---|---|
| *Point* | Source point |
| *X_Offset* | Offset in x dimension |
| *Y_Offset* | Offset in y dimension |

**DESCRIPTION**

Returns a copy of the source point translated by the specified offsets.

### TRANSLATE(RECTANGLE, real, real)

**SYNOPSIS**

```
RECTANGLE TRANSLATE(
      RECTANGLE Rect,
      real X_Offset,
      real Y_Offset)
```

**PARAMETERS**

| | |
|---|---|
| *Rect* | Source rectangle |
| *X_Offset* | Offset in x dimension |
| *Y_Offset* | Offset in y dimension |

**DESCRIPTION**

Returns a copy of the source rectangle translated by the specified offsets.

## VECTOR(POINT)

**SYNOPSIS**

```
VECTOR VECTOR(POINT Point)
```

**PARAMETERS**

| | |
|---|---|
| *Point* | Source point |

**DESCRIPTION**

Returns the origin vector of the source point.

## X(POINT)

**SYNOPSIS**

```
real X(POINT Point)
```

**PARAMETERS**

| | |
|---|---|
| *Point* | Source point |

**DESCRIPTION**

Returns the x coordinate of the source point.

## X(VECTOR)

**SYNOPSIS**

```
real X(VECTOR Vector)
```

**PARAMETERS**

| | |
|---|---|
| *Vector* | Source vector |

**DESCRIPTION**

Returns the x size of the source vector.

## Y(POINT)

**SYNOPSIS**

```
real Y(POINT Point)
```

**PARAMETERS**

| | |
|---|---|
| *Point* | Source point |

**DESCRIPTION**

Returns the y coordinate of the source point.

## Y(VECTOR)

**SYNOPSIS**

```
real Y(VECTOR Vector)
```

**PARAMETERS**

| | |
|---|---|
| *Vector* | Source vector |

**DESCRIPTION**

Returns the y size of the source vector.

# 5 Mathematical Functions

## ABSOLUTE(real)

**SYNOPSIS**

```
real ABSOLUTE(real Value)
```

**PARAMETERS**

| | |
|---|---|
| *Value* | Signed value |

**DESCRIPTION**

Returns the absolute value of the specified signed value.

## ABSOLUTE(integer)

**SYNOPSIS**

```
integer ABSOLUTE(integer Value)
```

**PARAMETERS**

| | |
|---|---|
| *Value* | Signed value |

**DESCRIPTION**

Returns the absolute value of the specified signed value.

## ARC_COSINE(real)

**SYNOPSIS**

```
real ARC_COSINE(real Value)
```

**PARAMETERS**

| | |
|---|---|
| *Value* | Cosine value |

**DESCRIPTION**

Returns the arc cosine of the specified cosine value.

## ARC_SINE(real)

**SYNOPSIS**

```
real ARC_SINE(real Value)
```

**PARAMETERS**

| | |
|---|---|
| *Value* | Sine value |

**DESCRIPTION**

Returns the arc sine of the specified sine value.

## ARC_TANGENT(real)

**SYNOPSIS**

```
real ARC_TANGENT(real Value)
```

**PARAMETERS**

| | |
|---|---|
| *Value* | Tangent value |

**DESCRIPTION**

Returns the arc tangent of the specified tangent value.

## COSINE(real)

**SYNOPSIS**

```
real COSINE(real Angle)
```

**PARAMETERS**

| | |
|---|---|
| *Angle* | Angle, in degrees |

**DESCRIPTION**

Returns the cosine of the specified angle.

## SINE(real)

**SYNOPSIS**

```
real SINE(real Angle)
```

**PARAMETERS**

| | |
|---|---|
| *Angle* | Angle, in degrees |

**DESCRIPTION**

Returns the sine of the specified angle.

## TANGENT(real)

**SYNOPSIS**

```
real TANGENT(real Angle)
```

**PARAMETERS**

| | |
|---|---|
| *Angle* | Angle, in degrees |

**DESCRIPTION**

Returns the tangent of the specified angle.

# 6 Image Processing Data Types

## BRUSH

### TYPE NAME

BRUSH

### CONSTRUCTOR FUNCTIONS

```
BRUSH BRUSH(
      FILE_NAME Segment
      [, real Brush_Width ]
      [, real Segment_Length ]
      [, real Maximum_Stroke_Length ]
      [, real Minimum_Angle ]
      [, real Maximum_Angle ])
BRUSH BRUSH(
      FILE_NAME Start_Cap,
      FILE_NAME Segment,
      FILE_NAME End_Cap
      [, real Brush_Width ]
      [, real Segment_Length ]
      [, real Maximum_Stroke_Length ]
      [, real Minimum_Angle ]
      [, real Maximum_Angle ])
```

### PARAMETERS

| | |
|---|---|
| *Segment* | Name of file containing brush stroke segment image (refer to Section 10.1 for format of file) |
| *Start_Cap* | Optional name of file containing brush stroke start cap image (refer to Section 10.1 for format of file); default value is the segment image |
| *End_Cap* | Optional name of file containing brush stroke end cap image (refer to Section 10.1 for format of file); default value is the segment image |
| *Brush_Width* | Optional brush width; default value is the height of segment image |
| *Segment_Length* | Optional segment length; default value is the width of segment image |
| *Maximum_Stroke_Length* | Optional maximum stroke length; default value is infinity |
| *Minimum_Angle* | Optional minimum brush rotation angle; default value is 0.0 |
| *Maximum_Angle* | Optional maximum brush rotation angle; default value is 360.0 |

### DESCRIPTION

A BRUSH defines the charactersitics of a brush used for stroking a path.

A BRUSH is constructed from a brush stroke segment image file, optional brush stroke start and end cap image files, and a number of other optional parameters.

The brush segment image is repeatedly stamped along the path to create the brush stroke. The optional start cap and end cap are stamped at the start and end of each open sub-path. The brush width defines the maximum distance between brush strokes laid down in parallel. The segment length defines the maximum spacing between one stamp and the next. The actual stamp spacing is adjusted according to the curvature of the path. The maximum stroke length defines the maximum length of a single stroke. A new stroke is started

each time the maximum stroke length is exceeded along a sub-path. The minimum and maximum brush rotation angles define the allowed rotation range for the brush. The rotation range allows pen-nib effects to be achieved with a brush.

## BUMP_MAP

### TYPE NAME

BUMP_MAP

### CONSTRUCTOR FUNCTIONS

BUMP_MAP **BUMP_MAP**(FILE_NAME *File_Name*)
BUMP_MAP **BUMP_MAP**(IMAGE *Image* [, CHANNEL *Channel* ])

### PARAMETERS

| | |
|---|---|
| *File_Name* | Name of file containing bump map (refer to Section 10.1 for format of file) |
| *Image* | Source image |
| *Channel* | Optional source channel |

### DESCRIPTION

A BUMP_MAP describes a two-dimensional array of surface heights. These heights are converted to slopes and hence surface normals which are used to compute the three-dimensional reflection of directional light from a surface.

Bump map height values range from 0.0 to 1.0. The maximum value of 1.0 maps to 0.1 model units. However, this can be scaled by a shadow depth (refer to the **ILLUMINATE** function).

A BUMP_MAP is constructed from a named bump map file, or from an image. In the latter case the bump map is either copied from a specified channel, or from the first channel if no channel is specified. The format of a bump map file is described in Section 10.1.

Bump map related functions are listed below. They are described in the next section.

### TESSELATION FUNCTIONS

BUMP_MAP **TESSELATE**(BUMP_MAP *Bump_Map*)

### TRANSFORMATION FUNCTIONS

BUMP_MAP **RESIZE**(BUMP_MAP *Bump_Map* [, VECTOR *Size* ])
BUMP_MAP **CROP**(BUMP_MAP *Bump_Map*)

## CHANNEL

### TYPE NAME

CHANNEL

### ENUMERATION

```
Lightness_Channel
A_Channel
B_Channel
Red_Channel
Green_Channel
Blue_Channel
Cyan_Channel
Magenta_Channel
Yellow_Channel
Hue_Channel
Saturation_Channel
Value_Channel
Luminance_Channel
```

### DESCRIPTION

A CHANNEL identifies a color channel of an image. Since the color space of an image may vary, a channel identifier appropriate to the color space of the image must be used in function calls which manipulate a channel.

## CLIP_IMAGE

### TYPE NAME

CLIP_IMAGE

### CONSTRUCTOR FUNCTIONS

```
CLIP_IMAGE CLIP_IMAGE(FILE_NAME File_Name)
CLIP_IMAGE CLIP_IMAGE(IMAGE Image, real Opacity)
CLIP_IMAGE CLIP_IMAGE(IMAGE Image, MATTE Matte)
```

### PARAMETERS

| | |
|---|---|
| *File_Name* | Name of file containing clip image (refer to Section 10.3 for format of file) |
| *Image* | Source image |
| *Matte* | Source matte |

### DESCRIPTION

A CLIP_IMAGE describes an image with an associated matte, intended to be positionally composited.

A clip image has a local coordinate space which has the same scale and orientation as the model space. The origin of the local coordinate space coincides with the axis point of the clip image. The clip image scales and rotates about its axis point. The axis point typically coincides with the geometric center of the clip image, but may be anywhere, including outside the bounding box of the clip image.

A CLIP_IMAGE is constructed from a named clip image file, or from an image and an opacity, or from an image and a matte. In the latter case the matte is cropped to the size of the image if necessary. The format of a clip image file is described in Section 10.3.

Clip image related functions are listed below. They are described in the next section.

### ATTRIBUTE FUNCTIONS

```
VECTOR SIZE(CLIP_IMAGE Clip_Image)
POINT AXIS_POINT(CLIP_IMAGE Clip_Image)
```

### SHADOW FUNCTIONS

```
CLIP_IMAGE SHADOW(
    CLIP_IMAGE Clip_Image,
    real Opacity)
```

### WARPING FUNCTIONS

```
CLIP_IMAGE WARP(
    CLIP_IMAGE Clip_Image,
    WARP_MAP Warp_Map
    [, boolean Maintain_Aspect ])
```

### TRANSFORMATION FUNCTIONS

```
CLIP_IMAGE RESIZE(CLIP_IMAGE Clip_Image [, VECTOR Size ])
CLIP_IMAGE CROP(CLIP_IMAGE Clip_Image)
CLIP_IMAGE SCALE(
    CLIP_IMAGE Clip_Image,
    real X_Scale,
    real Y_Scale)
CLIP_IMAGE SCALE(CLIP_IMAGE Clip_Image, real Scale)
CLIP_IMAGE SHEAR(
    CLIP_IMAGE Clip_Image,
    real X_Shear,
    real Y_Shear)
```

```
CLIP_IMAGE ROTATE(CLIP_IMAGE Clip_Image, real Angle)
CLIP_IMAGE TRANSFORM(
     CLIP_IMAGE Clip_Image,
     AFFINE_TRANSFORM Transform)
```

**TEXT SETTING AND RENDERING FUNCTIONS**

```
CLIP_IMAGE RENDER_TEXT(
     TEXT Text,
     FONT Font,
     COLOR Color
     [, TEXT_OPTIONS Options ])
CLIP_IMAGE RENDER_TEXT(
     TEXT Text,
     IMAGE_FONT Image_Font
     [, TEXT_OPTIONS Options ])
CLIP_IMAGE RENDER_TEXT(
     TEXT Text,
     IMAGE_FONT_3D Image_Font_3D
     [, TEXT_OPTIONS Options ])
```

## CLUT

**TYPE NAME**

CLUT

**CONSTRUCTOR FUNCTIONS**

```
CLUT CLUT(FILE_NAME File_Name)
```

**PARAMETERS**

| | |
|---|---|
| *File_Name* | Name of file containing color lookup table (refer to Section 10.4 for format of file) |

**DESCRIPTION**

A color lookup table (CLUT) describes an arbitrary and independent mapping function for each color channel. Each mapping function is encoded in a lookup table.

A CLUT is constructed from a named color lookup table file. The format of a color lookup table file is described in Section 10.4.

## CLUT_3D

**TYPE NAME**

CLUT_3D

**CONSTRUCTOR FUNCTIONS**

```
CLUT_3D CLUT_3D(FILE_NAME File_Name)
```

**PARAMETERS**

| | |
|---|---|
| *File_Name* | Name of file containing 3D color lookup table (refer to Section 10.5 for format of file) |

**DESCRIPTION**

A 3D color lookup table (CLUT_3D) describes an arbitrary color mapping function. The mapping function is encoded in a 3D lookup table.

A CLUT_3D is constructed from a named 3D color lookup table file. The format of a 3D color lookup table file is described in Section 10.5.

3D CLUT related procedures are listed below. They are described in the next section.

**OUTPUT PROCEDURES**

```
SAVE(CLUT_3D Table, FILE_NAME File_Name)
```

## COLOR

### TYPE NAME

COLOR

### CONSTRUCTOR FUNCTIONS

```
COLOR LAB_COLOR(real Lightness, real A, real B)
COLOR RGB_COLOR(real Red, real Green, real Blue)
COLOR CMY_COLOR(real Cyan, real Magenta, real Yellow)
COLOR HSV_COLOR(real Hue, real Saturation, real Value)
COLOR GRAYSCALE_COLOR(real Luminance)
COLOR BLACK()
COLOR WHITE()
```

### PARAMETERS

| | |
|---|---|
| *Lightness* | Lightness component of LAB color |
| *A* | A chrominance component of LAB color |
| *B* | B component of LAB color |
| *Red* | Red component of RGB color |
| *Green* | Green component of RGB color |
| *Blue* | Blue component of RGB color |
| *Cyan* | Cyan component of CMY color |
| *Magenta* | Magenta component of CMY color |
| *Yellow* | Yellow component of CMY color |
| *Hue* | Hue component of HSV color |
| *Saturation* | Saturation component of HSV color |
| *Value* | Value component of HSV color |
| *Luminance* | Luminance component of Grayscale color |

### DESCRIPTION

A COLOR describes a color in an LAB, RGB, CMY, HSV or Grayscale color space. The color is implicitly converted to the target color space of a particular function when used. A color is never used to specify the target color space of a function.

A COLOR is constructed from its components in the appropriate color space. **LAB_COLOR** constructs an LAB color. **RGB_COLOR** constructs an RGB color. **CMY_COLOR** constructs a CMY color. **HSV_COLOR** constructs an HSV color; and **GRAYSCALE_COLOR** constructs a grayscale color.

**BLACK** and **WHITE** construct black and white colors, respectively, in the working color space.

## COLOR_LIST

### TYPE NAME

COLOR_LIST

### DESCRIPTION

An alias for `string of` COLOR.

## COLOR_SPACE

### TYPE NAME

COLOR_SPACE

### ENUMERATION

```
LAB_Color_Space
RGB_Color_Space
CMY_Color_Space
HSV_Color_Space
Grayscale_Color_Space
```

**DESCRIPTION**

A COLOR_SPACE identifies a color space.

## DISPLACEMENT_MAP

**TYPE NAME**

DISPLACEMENT_MAP

**CONSTRUCTOR FUNCTIONS**

```
DISPLACEMENT_MAP DISPLACEMENT_MAP(FILE_NAME File_Name)
DISPLACEMENT_MAP DISPLACEMENT_MAP(IMAGE Image)
```

**PARAMETERS**

| | |
|---|---|
| *File_Name* | Name of file containing displacement map (refer to Section 10.4 for format of file) |
| *Image* | Source image |

**DESCRIPTION**

A DISPLACEMENT_MAP describes a two-dimensional array of x and y displacement factors, used to displace elements of an image or tiling (refer to the **ART_TILE** function).

The first channel of the map describes y displacement factors. The second channel describes x displacement factors. Displacement factors range from -1.0 to 1.0. Displacement factors are multiplied by a displacement range when used.

A DISPLACEMENT_MAP is constructed from a named displacement map file, or from the red and green channels of an RGB image. In the latter case the image color component range of 0.0 to 1.0 maps to a displacement factor range of −1.0 to 1.0. The format of a displacement map file is described in Section 10.4.

Displacement map related functions are listed below. They are described in the next section.

**TRANSFORMATION FUNCTIONS**

```
DISPLACEMENT_MAP RESIZE(
     DISPLACEMENT_MAP Displacement_Map
     [, VECTOR Size ])
DISPLACEMENT_MAP CROP(DISPLACEMENT_MAP Displacement_Map)
```

## DITHER_MATRIX

**TYPE NAME**

DITHER_MATRIX

**CONSTRUCTOR FUNCTIONS**

```
DITHER_MATRIX DITHER_MATRIX(FILE_NAME File_Name)
```

**PARAMETERS**

| | |
|---|---|
| *File_Name* | Name of file containing displacement map (refer to Section 10.7 for format of file) |

**DESCRIPTION**

A DITHER_MATRIX describes a two-dimensional array of intensity thresholds used for dithering an image (refer to the **DITHER** function). Since the intensity thresholds are compared directly with color component values, they have a range of 0.0 to 1.0.

A DITHER_MATRIX is constructed from a named dither matrix file. The format of a dither matrix file is described in Section 10.7.

## FACE

### TYPE NAME

FACE

### DATA MEMBERS

```
POINT Left_Eye
POINT Right_Eye
RECTANGLE Region
RECTANGLE Head_Region
real Orientation
```

### DESCRIPTION

A FACE describes a human or animal face detected in an image (refer to the **DETECT_DETAILED_FACE** and **DETECT_DETAILED_FACES** functions).

The **Left_Eye** and **Right_Eye** members describe the positions of the eyes. The **Region** member describes a tight-fitting region containing the face. The **Head_Region** member describes a loose-fitting region which contains the entire head, centered between the eyes . The **Orientation** member describes the orientation of the face, as an angle between 0.0 and 360.0 degrees.

## FACE_LIST

### TYPE NAME

FACE_LIST

### DESCRIPTION

An alias for `string of FACE`.

## FILE_NAME

### TYPE NAME

FILE_NAME

### DESCRIPTION

An alias for `string of character`.

## FONT

### TYPE NAME

FONT

### CONSTRUCTOR FUNCTIONS

```
FONT FONT(
     FONT_NAME Name
     [, FONT_STYLE Style ],
     real Size)
```

### PARAMETERS

| | |
|---|---|
| *Name* | Font name<br>(refer to Section 10.8 for formats) |
| *Style* | Optional font style |
| *Size* | Font size |

### DESCRIPTION

A FONT describes a collection of character glyph outlines and associated font metrics, used for setting and rendering text. The glyphs have a common design, style (bold, italic, etc.), and size.

A FONT is constructed from a named font known to the system, with a specified style and size. The size specifies the font size in model space, from the top of the font ascent, to the bottom of the font descent. Supported font formats are described in Section 10.8.

Font related functions are listed below. They are described in the next section.

**ATTRIBUTE FUNCTIONS**

```
real CHARACTER_ADVANCE(character Character, FONT Font)
```

## FONT_NAME

**TYPE NAME**

FONT_NAME

**DESCRIPTION**

An alias for `string of character`.

## FONT_STYLE

**TYPE NAME**

FONT_STYLE

**ENUMERATION**

```
Font_Style_Regular
Font_Style_Bold
Font_Style_Italic
Font_Style_BoldItalic
```

**DESCRIPTION**

A FONT_STYLE identifies a font style used during font creation.

## ILLUMINATION_OPTION

**TYPE NAME**

ILLUMINATION_OPTION

**CONSTRUCTOR FUNCTIONS**

```
ILLUMINATION_OPTION AMBIENT_REFLECTION(
     real Coefficient
     [, COLOR Color ])
ILLUMINATION_OPTION DIFFUSE_REFLECTION(real Coefficient)
ILLUMINATION_OPTION SPECULAR_REFLECTION(
     real Coefficient
     [, integer Exponent
     [, real Color_Coefficient ] ])
ILLUMINATION_OPTION SHADOW_DEPTH(real Depth)
ILLUMINATION_OPTION NO_LIGHT_FALLOFF()
ILLUMINATION_OPTION LINEAR_LIGHT_FALLOFF()
ILLUMINATION_OPTION SQUARED_LIGHT_FALLOFF()
```

**PARAMETERS**

| | |
|---|---|
| *Coefficient* | Ambient, diffuse or specular reflection coefficient |
| *Color* | Optional ambient color; default value is white |
| *Exponent* | Optional specular exponent; default value is 5 |
| *Color_Coefficient* | Optional specular color coefficient; default value is 0.0 |
| *Depth* | Shadow depth |

**DESCRIPTION**

An ILLUMINATION_OPTION describes an optional surface reflection characteristic, used during lighting calculations (refer to the **ILLUMINATE** function).

**AMBIENT_REFLECTION** creates an ambient reflection illumination option with the specified ambient reflection coefficient and ambient color. The coefficient specifies the amount of *non-directional ambient light* reflected by the surface, from 0.0 for none to 1.0 for all. The color specifies the color of the ambient light. The default lighting behaviour, in the absence of an ambient reflection illumination option, is an ambient reflection coefficient of 0.0.

**DIFFUSE_REFLECTION** creates a diffuse reflection illumination option with the specified diffuse reflection coefficient. The coefficient specifies the amount of *directional light reflected non-directionally* by the surface, from 0.0 for none to 1.0 for all. Directional light comes from one or more light sources (refer to the LIGHT data type). The default lighting behaviour, in the absence of a diffuse reflection illumination option, is a diffuse reflection coefficient of 1.0.

**SPECULAR_REFLECTION** creates a specular reflection illumination option with the specified specular reflection coefficient, exponent and color coefficient. The coefficient specifies the amount of *directional light reflected directionally* by the surface, from 0.0 for none to 1.0 for all. Directional light comes from one or more light sources (refer to the LIGHT data type). The exponent specifies the spread of the specularly reflected light. A large exponent specifies a small spread, and hence small specular highlights. The color coefficient specifies the color of the specularly reflected light, from 0.0 for the light color to 1.0 for the surface color, with interpolation in-between. The default lighting behaviour, in the absence of a specular reflection illumination option, is a specular reflection coefficient of 0.0.

**SHADOW_DEPTH** creates a shadow depth illumination option with the specified shadow depth. The shadow depth specifies a scaling factor on the bump map which defines the surface texture. The default lighting behaviour, in the absence of a shadow depth illumination option, is that the bump map is not scaled.

**NO_LIGHT_FALLOFF** creates an illumination option which specifies no falloff in the intensity of light from a light source with increasing distance from the light source. **LINEAR_LIGHT_FALLOFF** creates an illumination option which specifies linear falloff with increasing distance. **SQUARED_LIGHT_FALLOFF** creates an illumination option which specifies squared falloff with increasing distance. The default lighting behaviour, in the absence of a light falloff illumination option, is squared falloff.

The reflection coefficients are normally in the range 0.0 to 1.0, but may be larger than 1.0 for more extreme lighting.

## ILLUMINATION_OPTIONS

### TYPE NAME

ILLUMINATION_OPTIONS

### DESCRIPTION

An alias for `string of ILLUMINATION_OPTION`.

## IMAGE

### TYPE NAME

IMAGE

### CONSTRUCTOR FUNCTIONS

```
IMAGE IMAGE(FILE_NAME File_Name)
IMAGE IMAGE(COLOR Color [, COLOR_SPACE Color_Space ])
IMAGE IMAGE(MATTE Matte)
IMAGE IMAGE(CLIP_IMAGE Clip_Image)
IMAGE IMAGE(IMAGE Image, BUMP_MAP Bump_Map)
```

**PARAMETERS**

| | |
|---|---|
| *File_Name* | The name of a file containing an image (refer to Section 10.9 for format of file) |
| *Color* | Flat image color |
| *Color_Space* | Optional target color space; default value is the working color space |
| *Image* | Optional source image for size |
| *Matte* | Source matte |
| *Clip_Image* | Source clip image |
| *Bump_Map* | Source bump map |

**DESCRIPTION**

An IMAGE describes a two-dimensional array of color pixels.

An image has a local coordinate space which has the same scale and orientation as the model space. The origin of the local coordinate space coincides with the axis point of the image. The image scales and rotates about its axis point. The axis point always coincides with the geometric center of the image.

The natural image color space is the working color space. However, an image can be converted to and from other color spaces, such as HSV and grayscale, for specific effects.

An image may contain an optional bump map channel, e.g. when created by artistic tiling (refer to the **ART_TILE** functions). The bump map channel is used during lighting calculations (refer to the **ILLUMINATE** function).

An image is constructed from a named image file, or from a flat color, or from the opacity channel of a matte (resulting in a Grayscale image), or from the color channels of a clip image, or from an image and a bump map. In the flat color case, the size of the image is the working image size, and the color space is the specified color space, or the working color space in the absence of a color space. The format of an image file is described in Section 10.9.

Image related functions are listed below. They are described in the next section.

**ATTRIBUTE FUNCTIONS**

```
VECTOR SIZE(IMAGE Image)
COLOR_SPACE COLOR_SPACE(IMAGE Image)
```

**INPUT FUNCTIONS**

```
IMAGE PHOTO()
IMAGE RAW_PHOTO()
```

**OUTPUT PROCEDURES**

```
PRINT(IMAGE Image)
SAVE(IMAGE Image, FILE_NAME File_Name)
```

**COLOR MAPPING FUNCTIONS**

```
IMAGE LOOKUP_COLOR(IMAGE Image, CLUT Table)
IMAGE LOOKUP_COLOR(IMAGE Image, CLUT_3D Table)
IMAGE CONVERT_COLOR_SPACE(
     IMAGE Image,
     COLOR_SPACE Color_Space)
IMAGE DESATURATE(IMAGE Image)
IMAGE INVERT(IMAGE Image)
IMAGE POSTERIZE(IMAGE Image, integer Levels)
IMAGE THRESHOLD(IMAGE Image, real Threshold)
IMAGE AREA_THRESHOLD(IMAGE Image, real Proportion)
```

**COLOR QUANTIZATION FUNCTIONS**

```
IMAGE QUANTIZE(IMAGE Image, PALETTE Palette)
IMAGE QUANTIZE_HUE(IMAGE Image, integer Levels)
```

**COLOR DITHERING FUNCTIONS**

```
IMAGE DITHER(
```

```
        IMAGE Image,
        DITHER_MATRIX Matrix
        [, COLOR_LIST Color_List ])
```

### COLOR BLENDING FUNCTIONS

```
IMAGE HORIZONTAL_COLOR_BLEND(
     COLOR Start_Color,
     COLOR End_Color
     [, COLOR_SPACE Color_Space ])
IMAGE VERTICAL_COLOR_BLEND(
     COLOR Start_Color,
     COLOR End_Color
     [, COLOR_SPACE Color_Space ])

IMAGE COLOR_BLEND(
     COLOR Color1,
     COLOR Color2,
     COLOR Color3,
     COLOR Color4
     [, COLOR_SPACE Color_Space ])
```

### NOISE FUNCTIONS

```
IMAGE ADD_NOISE(IMAGE Image, real Amount)
IMAGE ADD_MONOCHROME_NOISE(
     IMAGE Image,
     real Amount)
```

### COMPOSITING FUNCTIONS AND PROCEDURES

```
IMAGE COMPOSITE(
     IMAGE Image,
     IMAGE Matte,
     IMAGE Background_Image
     [, real Height ])
IMAGE COMPOSITE(
     CLIP_IMAGE Clip_Image,
     IMAGE Background_Image,
     POINT Position
     [, real Height ])
COMPOSITE(
     CLIP_IMAGE Clip_Image,
     variable IMAGE Background_Image,
     POINT Position
     [, real Height ])
IMAGE ADD(IMAGE Image, COLOR Color)
IMAGE ADD(IMAGE Image1, IMAGE Image2)
IMAGE SUBTRACT(IMAGE Image, COLOR Color)
IMAGE SUBTRACT(IMAGE Image1, IMAGE Image2)
IMAGE MULTIPLY(IMAGE Image, COLOR Color)
IMAGE MULTIPLY(IMAGE Image1, IMAGE Image2)
IMAGE MIN(IMAGE Image1, IMAGE Image2)
IMAGE MAX(IMAGE Image1, IMAGE Image2)
```

### FILTERING FUNCTIONS

```
IMAGE BLUR(IMAGE Image, real Radius)
IMAGE CONVOLVE(
     IMAGE Image,
     KERNEL Kernel
     [, boolean Absolute_Value ])
IMAGE FIND_EDGES(IMAGE Image)
IMAGE MEDIAN_FILTER(IMAGE Image)
IMAGE MIN_FILTER(IMAGE Image [, integer Radius ])
IMAGE MAX_FILTER(IMAGE Image [, integer Radius ])
IMAGE MEAN_MIN_MAX_FILTER(IMAGE Image [, integer Radius ])
IMAGE SKELETONIZE(IMAGE Image)
```

```
PATH SKELETONIZE_TO_PATH(IMAGE Image)
```

**CHANNEL MANIPULATION FUNCTIONS**

```
IMAGE EXTRACT_CHANNEL(
      IMAGE Image,
      CHANNEL Channel)
IMAGE REPLACE_CHANNEL(
      IMAGE Image,
      CHANNEL Channel,
      IMAGE Source_Image)
IMAGE REPLACE_CHANNEL(
      IMAGE Image,
      CHANNEL Channel,
      real Value)
```

**TRANSFORMATION FUNCTIONS**

```
IMAGE SCALE(IMAGE Image, real X_Scale, real Y_Scale)
IMAGE SCALE(real Scale, IMAGE Image)
IMAGE SHEAR(IMAGE Image, real X_Shear, real Y_Shear)
IMAGE RESIZE(IMAGE Image [, VECTOR Size ])
IMAGE FLIP_VERTICALLY(IMAGE Image)
IMAGE FLIP_HORIZONTALLY(IMAGE Image)
IMAGE ROTATE(IMAGE Image, real Angle)
IMAGE TRANSLATE(IMAGE Image, real X_Offset, real Y_Offset)
IMAGE TRANSFORM(IMAGE Image, AFFINE_TRANSFORM Transform)
IMAGE CROP(IMAGE Image)
IMAGE CROP(IMAGE Image, real X_Size, real Y_Size)
IMAGE CROP(IMAGE Image, RECTANGLE Region)
```

**TESSELATION FUNCTIONS**

```
IMAGE TESSELATE(IMAGE Image)
```

**ARTISTIC TILING FUNCTIONS**

```
IMAGE ART_TILE(
      IMAGE Image,
      TILE_PATTERN Tile_Pattern
      [, TILE_OPTIONS Options ])
IMAGE ART_TILE(
      IMAGE Image,
      MATTE Layer_Selection,
      TILE_LAYER_LIST Layer_List
      [, TILE_OPTIONS Options ])
```

**PAINTING PROCEDURES**

```
STROKE(
      PATH Path,
      BRUSH Brush,
      COLOR Color,
      variable IMAGE Image)
STROKE(
      PATH Path,
      BRUSH Brush,
      PAINT Paint,
      variable IMAGE Image)
```

**WARPING FUNCTIONS**

```
IMAGE WARP(
      IMAGE Image,
      WARP_MAP Warp_Map
      [, boolean Maintain_Aspect ]
      [, RECTANGLE_LIST Regions ])
```

**ILLUMINATION FUNCTIONS**

```
IMAGE ILLUMINATE(
      IMAGE Image,
```

```
        ILLUMINATION_OPTIONS Options,
        LIGHT_LIST Light_List
        [, BUMP_MAP Bump_Map ])
```

**FACE DETECTION FUNCTIONS**

```
FACE DETECT_DETAILED_FACE(IMAGE Image)
FACE_LIST DETECT_DETAILED_FACES(
     IMAGE Image
     [, integer Maximum_Faces
     [, real Face_Threshold ]])
RECTANGLE DETECT_FACE(IMAGE Image)
RECTANGLE_LIST DETECT_FACES(
     IMAGE Image
     [, integer Maximum_Faces
     [, real Face_Threshold ]])
```

## IMAGE_FONT

**TYPE NAME**

IMAGE_FONT

**CONSTRUCTOR FUNCTIONS**

```
IMAGE_FONT IMAGE_FONT(FILE_NAME File_Name, real Size)
```

**PARAMETERS**

| | |
|---|---|
| *File_Name* | Name of file containing image font (refer to Section 10.10 for format of file) |
| *Size* | Font size |

**DESCRIPTION**

An IMAGE_FONT describes a collection of character glyph images and associated font metrics, used for setting and rendering text. The glyphs have a common design, style (bold, italic, etc.), and size.

An IMAGE_FONT is constructed by loading it from a named image font file, at a specified size. The size specifies the font size in model space, from the top of the font ascent, to the bottom of the font descent. The format of an image font file is described in Section 10.10.

Image font related functions are listed below. They are described in the next section.

**ATTRIBUTE FUNCTIONS**

```
real CHARACTER_ADVANCE(
     character Character,
     IMAGE_FONT Image_Font)
```

## IMAGE_FONT_3D

**TYPE NAME**

IMAGE_FONT_3D

**CONSTRUCTOR FUNCTIONS**

```
IMAGE_FONT_3D IMAGE_FONT_3D(FILE_NAME File_Name, real Size)
```

**PARAMETERS**

| | |
|---|---|
| *File_Name* | Name of file containing 3D image font (refer to Section 10.11 for format of file) |
| *Size* | Font size |

**DESCRIPTION**

An IMAGE_FONT_3D describes a collection of character glyph images, rendered in three dimensions from a fixed viewpoint, and associated font metrics, used for setting and rendering text in pseudo-3D. The glyphs have a common design, style (bold, italic, etc.), and size.

An IMAGE_FONT_3D is constructed by loading it from a named 3D image font file, at a specified size. The size specifies the font size in model space, from the top of the font ascent, to the bottom of the font descent. The format of a 3D image font file is described in Section 10.11.

## KERNEL

### TYPE NAME

KERNEL

### CONSTRUCTOR FUNCTIONS

```
KERNEL KERNEL(FILE_NAME File_Name)
KERNEL KERNEL(string of string of real Values)
KERNEL KERNEL(
     string of string of integer Values
     [, integer Scale ])
```

### PARAMETERS

| | |
|---|---|
| *File_Name* | Name of file containing kernel (refere to Section 10.12 for format of file) |
| *Values* | Matrix of real or integer kernel values |
| *Scale* | Optional scale factor for integer kernel values; default value is the number of non-zero kernel values |

### DESCRIPTION

A KERNEL describes an arbitrary convolution kernel, used to convolve an image (refer to the CONVOLVE function).

A KERNEL is constructed by loading it from a named kernel file, or by building it from a matrix or values directly. In the latter case the values specify real or integer coefficients. The default scale factor for both a real and integer value matrix is the number of non-zero values in the matrix. For integer values, an explicit scale factor may be specified. The format of a kernel file is described in Section 10.12.

Kernel related procedures are listed below. They are described in the next section.

### EXAMPLES

Low-pass filter kernel:

```
KERNEL
([
    [1.0, 1.0, 1.0],
    [1.0, 1.0, 1.0],
    [1.0, 1.0, 1.0]
])
```

High-pass filter kernel:

```
KERNEL
([
    [-1, -1, -1],
    [-1,  8, -1],
    [-1, -1, -1],
    9
])
```

### OUTPUT PROCEDURES

```
SAVE(KERNEL Kernel, FILE_NAME File_Name)
```

## LIGHT

### TYPE NAME

LIGHT

### CONSTRUCTOR FUNCTIONS

```
LIGHT LIGHT(
      COLOR Color,
      real Direction,
      real Tilt
      [, real Distance
      [, POINT Target
      [, real Cone
      [, real Penumbra ] ] ] ])
```

### PARAMETERS

| | |
|---|---|
| *Color* | Color of light |
| *Direction* | Direction angle, in degrees about z axis |
| *Tilt* | Tilt angle, in degrees |
| *Distance* | Optional distance to light source; default value is infinity |
| *Target* | Optional target point; default value is the origin |
| *Cone* | Optional half-angle of light cone, in degrees; default is no cone |
| *Penumbra* | Optional angle between penumbra cone and light cone; default value is 0.05 × cone half-angle |

### DESCRIPTION

A LIGHT describes a directional light source, used during lighting calculations (refer to the **ILLUMINATE** function).

A LIGHT is constructed from a color, a direction, and a tilt. The color specifies the color of the light. The direction specifies the direction in which the light lies, i.e. the direction from the target point to the light. The tilt specifies the tilt of the light source, i.e. the angle it makes with the image plane.

An optional distance specifies the distance of the light source from the target point in the image plane. When the distance is specified, the light attenuates with the square of the distance. When the distance defaults to infinity, the light does not attenuate. The attenuation factor is normalised so that it is 1.0 at the closest point in the image plane to the light source.

An optional target point specifies the point in the image plane at which the light source points. The default target point is the origin – i.e. the center of the image.

An optional cone half-angle specifies the shape of a conical, i.e. shaded, light source. The default, when no cone half-angle is specified, is that the light source is omni-directional.

An optional penumbra angle specifies the angle between the outside of the penumbra cone and the light cone. The light attenuates within the penumbra region in pseudo-relation to the area of the light source visible. The default angle between the penumbra cone and light cone is 0.05 times the cone half-angle.

## LIGHT_LIST

### TYPE NAME

LIGHT_LIST

### DESCRIPTION

An alias for `string of LIGHT`.

## MATTE

### TYPE NAME

MATTE

### CONSTRUCTOR FUNCTIONS

```
MATTE MATTE(FILE_NAME File_Name)
```

```
MATTE MATTE(real Opacity)
MATTE MATTE(IMAGE Image, CHANNEL Channel)
MATTE MATTE(CLIP_IMAGE Clip_Image)
```

**PARAMETERS**

| | |
|---|---|
| *File_Name* | Name of file containing matte (refer to Section 10.13 for format of file) |
| *Opacity* | Flat matte opacity |
| *Image* | Source image |
| *Channel* | Source channel |
| *Clip_Image* | Source clip image |

**DESCRIPTION**

A MATTE describes a two-dimensional array of opacities, used for compositing a foreground image with a background image (refer to the **COMPOSITE** function).

Opacity values range from 0.0 (fully transparent) to 1.0 (fully opaque).

A matte is constructed from a named matte file, or from a flat opacity, or from a specified channel of an image, or from the matte channel of a clip image. The format of a matte file is described in Section 10.13.

Matte related functions are listed below. They are described in the next section.

**COMPOSITING FUNCTIONS**

```
MATTE MULTIPLY(MATTE Matte, real Opacity)
```

**TRANSFORMATION FUNCTIONS**

```
MATTE RESIZE(MATTE Matte [, VECTOR Size ])
MATTE CROP(MATTE Matte)
MATTE TRANSFORM(MATTE Matte, AFFINE_TRANSFORM Transform)
```

## PAINT

**TYPE NAME**

```
PAINT
```

**CONSTRUCTOR FUNCTIONS**

```
PAINT PAINT(
      [ COLOR Color
      [, real Stiffness
      [, real Minimum_Thickness
      [, boolean Scalable_Opacity ]]]])
```

**PARAMETERS**

| | |
|---|---|
| *Color* | Optional paint color; default value is black |
| *Stiffness* | Optional paint stiffness; default value is 0.0 |
| *Minimum_Thickness* | Optional minimum paint thickness; default value is 0.0 |
| *Scalable_Opacity* | Optional scalable opacity flag; default value is `false` |

**DESCRIPTION**

A PAINT describes the characteristics of a paint applied to a brush stroke during path stroking (refer to the **STROKE** functions).

A PAINT is constructed from a color and a number of other optional parameters which control how a brush stroke painted with the paint interacts with the underlying surface texture.

The paint stiffness specifies to what extent the brush stroke obscures the underlying surface texture. The minimum paint thickness specifies a minimum thickness with which the brush stroke covers background surface texture which protrudes through the brush stroke. The scalable opacity flag specifies whether the brush stroke opacity is scaled down where background surface texture protrudes into the brush stroke.

## PALETTE

### TYPE NAME

PALETTE

### CONSTRUCTOR FUNCTIONS

PALETTE **PALETTE**(FILE_NAME *File_Name*)
PALETTE **PALETTE**(COLOR_LIST *Color_List*)

### PARAMETERS

| | |
|---|---|
| *File_Name* | Name of a file containing a palette (refer to Section 10.14 for format of file) |
| *Color_List* | List of source colors for palette |

### DESCRIPTION

A PALETTE describes a discrete set of RGB colors, used to quantize an image (refer to the **QUANTIZE** function).

A PALETTE is constructed from a named palette file, or from a list of colors. In the latter case the colors must be RGB colors. The format of a palette file is described in Section 10.14.

Palette related procedures are listed below. They are described in the next section.

### OUTPUT PROCEDURES

**SAVE**(PALETTE *Palette*, FILE_NAME *File_Name*)

## TEXT

### TYPE NAME

TEXT

### DESCRIPTION

An alias for `string of character`.

## TEXT_OPTION

### TYPE NAME

TEXT_OPTION

### CONSTRUCTOR FUNCTIONS

TEXT_OPTION **TEXT_ALIGN_LEFT**()
TEXT_OPTION **TEXT_CENTER**()
TEXT_OPTION **TEXT_ALIGN_RIGHT**()
TEXT_OPTION **TEXT_JUSTIFY**()
TEXT_OPTION **TEXT_ALIGN_TOP**()
TEXT_OPTION **TEXT_CENTER_VERTICALLY**()
TEXT_OPTION **TEXT_ALIGN_BOTTOM**()
TEXT_OPTION **TEXT_JUSTIFY_VERTICALLY**()
TEXT_OPTION **TEXT_SET_TO_REGION**(VECTOR *Region_Size*)
TEXT_OPTION **TEXT_FIT_TO_REGION**(VECTOR *Region_Size*)
TEXT_OPTION **TEXT_TRACKING**(real *Tracking*)
TEXT_OPTION **TEXT_LEADING**(real *Leading*)

### PARAMETERS

| | |
|---|---|
| *Region_Size* | Size of target region |
| *Tracking* | Inter-character tracking value |
| *Leading* | Inter-line leading value |

### DESCRIPTION

A TEXT_OPTION describes an optional text formatting control, used during text setting (refer to the **SET_TEXT** function).

**TEXT_ALIGN_LEFT**, **TEXT_CENTER**, **TEXT_ALIGN_RIGHT** and **TEXT_JUSTIFY** create horizontal alignment text options which specify left alignment, horizontal centering, right alignment, and full horizontal justification, respectively. The default text setting behaviour, in the absence of any horizontal alignment option, is left alignment.

**TEXT_ALIGN_TOP**, **TEXT_CENTER_VERTICALLY**, **TEXT_ALIGN_BOTTOM** and **TEXT_JUSTIFY_VERTICALLY** create vertical alignment text options which specify top alignment, vertical centering, bottom alignment, and full vertical justification, respectively. The default text setting behaviour, in the absence of any vertical alignment option, is top alignment.

**TEXT_SET_TO_REGION** creates a text option which specifies that the text be set to the specifies region. Excess text is discarded. **TEXT_FIT_TO_REGION** creates a text option which specifies that the text be fitted to the specified region. The font size is scaled so that the text fits the region, so that no text is discarded. The default text setting behaviour, in the absence of any region text option, is an infinite region. When the region is infinite, only explicit newline characters in the text lead to text wrapping.

**TEXT_TRACKING** creates a text option which specifies an inter-character tracking value. **TEXT_LEADING** creates a text option which specifies an inter-line leading value. The default text setting behaviour, in the absence of tracking or leading text options, is no tracking or leading. The leading and tracking values specify a proportion of the font height.

## TEXT_OPTIONS

### TYPE NAME

TEXT_OPTIONS

### DESCRIPTION

An alias for `string of TEXT_OPTION`.

## TILE

### TYPE NAME

TILE

### CONSTRUCTOR FUNCTIONS

TILE **TILE**(FILE_NAME *File_Name* [, VECTOR *Offset* ])
TILE **TILE**(IMAGE *Image*)

### PARAMETERS

| | |
|---|---|
| *File_Name* | Name of file containing tile (refer to Section 10.16 for format of file) |
| *Offset* | Optional inter-tile offset |
| *Image* | Source image |

### DESCRIPTION

A TILE describes a tile shape, a tile surface texture, and an inter-tile offset. A tile is used singly or as part of a pattern to artistically tile an image (refer to the TILE_PATTERN data type and the **ART_TILE** functions).

When the tile participates in a tile pattern, the inter-tile offset specifies the offset from the tile to the next tile in the pattern. The inter-tile offset is specified in *tile pixel units*, rather than model space units, since a tile is normally carefully designed at the pixel level to form part of a tile pattern.

The tile shape is defined by a matte channel. The tile surface texture is defined by a bump map channel.

A TILE is constructed from a named tile file, or from an RGB image. In the former case the optional offset specifies the tile's inter-tile offset. In the latter case the matte and sur-

face texture channels are copied from the red and green channels of the image, and the tile's inter-tile offset is set to the default. The default horizontal inter-tile offset is the width of the tile, and the default vertical inter-tile offset is 0.0. The format of a tile file is described in Section 10.16.

## TILE_LAYER

### TYPE NAME

TILE_LAYER

### CONSTRUCTOR FUNCTIONS

```
TILE_LAYER TILE_LAYER(
     TILE_PATTERN Tile_Pattern,
     real Threshold)
```

### PARAMETERS

| | |
|---|---|
| *Tile_Pattern* | Tile pattern for layer |
| *Threshold* | Layer threshold |

### DESCRIPTION

A TILE_LAYER describes the tile pattern and threshold of one layer of a multi-layer tiling (refer to the **ART_TILE** functions). The threshold specifies the value, in the related layer selection matte, at which the layer becomes active.

A TILE_LAYER is constructed from a tile pattern and a threshold value between 0.0 and 1.0.

## TILE_LAYER_LIST

### TYPE NAME

TILE_LAYER_LIST

### DESCRIPTION

An alias for `string of TILE_LAYER`.

## TILE_LIST

### TYPE NAME

TILE_LIST

### DESCRIPTION

An alias for `string of TILE`.

## TILE_OPTION

### TYPE NAME

TILE_OPTION

### CONSTRUCTOR FUNCTIONS

```
TILE_OPTION TILE_AVERAGE_COLOR()
TILE_OPTION TILE_MAP_COLOR_TO_PALETTE(PALETTE Palette)
TILE_OPTION TILE_DIFFUSE_COLOR_ERROR()
TILE_OPTION TILE_DIFFUSE_LAYER_ERROR()
TILE_OPTION TILE_JITTER_COLOR(real Color_Range)
TILE_OPTION TILE_INVERT_CHROMA(real Inversion_Rate)
TILE_OPTION TILE_JITTER_POSITION(VECTOR Position_Range)
TILE_OPTION TILE_DISPLACE_POSITION(
     DISPLACEMENT_MAP Displacement_Map,
     VECTOR Displacement_Range)
TILE_OPTION TILE_GENERATE_BUMP([ real Tile_Bump_Scale ])
TILE_OPTION TILE_SUM_BUMPS()
```

```
TILE_OPTION TILE_AVERAGE_BUMP(
     [ integer Minimum_Tile_Thickness ])
TILE_OPTION TILE_SCALE_MATTE_BY_BUMP()
TILE_OPTION TILE_OVER_BACKGROUND(IMAGE Image)
TILE_OPTION TILE_ORIENT_TILES(
     VECTOR_MAP Vector_Map
     [, real Minimum_Angle,
     real Maximum_Angle ])
TILE_OPTION TILE_RANDOMIZE_LAYERING()
TILE_OPTION TILE_AVOID_FOOTPRINTS()
```

### PARAMETERS

| | |
|---|---|
| *Palette* | Target palette |
| *Color_Range* | Color jitter range |
| *Inversion_Rate* | Color inversion rate |
| *Position_Range* | Position jitter range |
| *Displacement_Map* | Displacement map |
| *Displacement_Range* | Displacement range |
| *Tile_Bump_Scale* | Optional tile bump map scale factor; default value is 1.0 |
| *Minimum_Tile_Thickness* | Optional minimum tile thickness; default value is 0.0 |
| *Image* | Background image |
| *Vector_Map* | Vector map giving tile rotation |
| *Minimum_Angle* | Optional minimum tile rotation angle; default value is 0.0 |
| *Maximum_Angle* | Optional maximum tile rotation angle; default value is 360.0 |

### DESCRIPTION

A TILE_OPTION describes an optional tiling control, used during artistic image tiling (refer to the **ART_TILE** functions).

**TILE_AVERAGE_COLOR** creates a tile option which specifies that the tile color be computed from the average of the image pixels covered by the tile. The default tiling behaviour, in the absence of this option, it that the pixel-by-pixel tile color is the same as the pixel-by-pixel color of the region of the image it covers.

**TILE_MAP_COLOR_TO_PALETTE** creates a tile option which specifies that the averaged tile color be mapped to the specified palette. This option is only effective if the tile color is being averaged.

**TILE_DIFFUSE_COLOR_ERROR** creates a tile option which specifies that the error between the palette-mapped tile color and the actual (averaged) tile color be diffused into neighboring tiles. This has the effect of minimising the error between the overall tiling color and the overall image color. This option is only effective if the tile color is being averaged and mapped to a palette.

During multi-layer tiling, a per-layer threshold value is compared with a layer-selection matte. The values in the layer-selection matte covered by a tile are always averaged and then thresholded to 0.0 or 1.0. **TILE_DIFFUSE_LAYER_ERROR** creates a tile option which specifies that the error between the thresholded tile layer selection value and the actual (averaged) tile layer selection value be diffused into neighboring tiles. This has the effect of minimising the error between the overall layer "opacity" (i.e. the relative tile coverage) and the overall matte opacity. This option is only effective during multi-layer tiling.

**TILE_JITTER_COLOR** creates a tile option which specifies that the averaged tile color be jittered randomly within the specified range (i.e. ± *range* / 2). This option is only effective if the tile color is being averaged.

**TILE_INVERT_CHROMA** creates a tile option which specifies that the chroma components of the averaged tile color be randomly inverted at the specified rate. This option is only effective if the tile color is being averaged.

**TILE_JITTER_POSITION** creates a tile option which specifies that the tile position be jittered within the specified x and y range (i.e. ± *range* / 2).

**TILE_DISPLACE_POSITION** creates a tile option which specifies that the tile position be displaced according to the specified displacement map. During tiling, the x and y displacement factors in the map, which are in the range -1.0 to 1.0, are scaled by the specified displacement range (i.e. to yield ± *range*).

**TILE_GENERATE_BUMP** creates a tile option which specifies that a bump map channel be generated in the tiled image. The bump map channel is used during lighting calculations (refer to the **ILLUMINATE** function). The optional scale factor specifies a factor by which the tile bump maps are scaled before being used. The default scale factor is 1.0.

**TILE_SUM_BUMPS** creates a tile option which specifies that the bump map of the background image be summed with the bump map of the tile. The default tiling behaviour, in the absence of this option, is that the bump map of the tile simply replaces the bump map of the region of the background image it covers. This option is only effective if a bump map channel is being generated.

**TILE_AVERAGE_BUMP** creates a tile option which specifies that the bump map of the background image covered by the tile be averaged before being combined with the bump map of the tile. This option is only effective if a bump map channel is being generated and background and tile bump maps are being summed.

In the process of combining the background bump map with the tile bump map, the tile bump map may thinned. **TILE_SCALE_MATTE_BY_BUMP** creates a tile option which specifies that the matte of the tile be scaled by the degree of thinning. This has the effect of translating thinning into greater transparency. This option is only effective if a bump map channel is being generated and background and tile bump maps are being summed.

The bump map related options are described in more detail in the discussion of artistic tiling in Section 2.1.

**TILE_OVER_BACKGROUND** creates a tile option which specifies that the tiling be combined with the specified background image. If a bump map is being generated, then the specified background image must also contain a bump map channel.

**TILE_ORIENT_TILES** creates a tile option which specifies that tiles be rotated according to the direction in the specified vector map. The optional angle range specifies the range of rotations allowed for the tiles. Angles outside the range are clipped to the range. The default range is 0.0 to 360.0.

**TILE_RANDOMIZE_LAYERING** creates a tile option which specifies that tiles be composited with the background in random order. This is useful if the tiles overlap. The default behaviour, in the absence of this option, is that tiles are composited left-to-right, top-to-bottom.

## TILE_OPTIONS

**TYPE NAME**

TILE_OPTIONS

**DESCRIPTION**

An alias for `string of TILE_OPTION`.

## TILE_PATTERN

**TYPE NAME**

TILE_PATTERN

**CONSTRUCTOR FUNCTIONS**

TILE_PATTERN **TILE_PATTERN**(

```
        [ VECTOR Start, ]
        VECTOR Offset,
        TILE_LIST Tile_List)
TILE_PATTERN RANDOM_TILE_PATTERN(
        [ VECTOR Start, ]
        VECTOR Offset,
        TILE_LIST Tile_List)
```

**PARAMETERS**

| | |
|---|---|
| *Start* | Offset to starting point in pattern |
| *Offset* | Offset between one pattern placement and the next |
| *Tile_List* | List of tiles in pattern |

**DESCRIPTION**

A TILE_PATTERN describes a set of tiles which together make up a tile pattern, together with a starting offset, which defines an aesthetically pleasing starting point for the tiling, and an inter-pattern offset, which defines how to place one pattern after another to fully tile a plane. A tile pattern is used to artistically tile an image (refer to the **ART_TILE** functions).

A tile pattern is optionally random – i.e. the tiles which make up the pattern do not have a fixed relationship, but are instead designed to be placed in random order. A random tile pattern is used to render an image in a more of a painting than a tiling style, as described in Section 2.1.

A TILE_PATTERN is constructed from an optional starting offset, an inter-pattern offset, and a list of tiles. The tiles themselves describe the required inter-tile (intra-pattern) offsets required for the tiling (refer to the TILE data type). A random TILE_PATTERN is constructed in the same way, but using the **RANDOM_TILE_PATTERN** constructor.

## VECTOR_MAP

**TYPE NAME**

VECTOR_MAP

**CONSTRUCTOR FUNCTIONS**

```
VECTOR_MAP VECTOR_MAP(FILE_NAME File_Name)
```

**PARAMETERS**

| | |
|---|---|
| *File_Name* | Name of file containing vector map<br>(refer to Section 10.1 for format of file) |

**DESCRIPTION**

A VECTOR_MAP describes a two-dimensional array of vectors, each consisting of a direction angle in the range 0.0 to 360.0 degrees, and a magnitude in the range 0.0 to 1.0.

A VECTOR_MAP is constructed from a named vector map file. The format of a vector map file is described in Section 10.1.

Vector map related functions are listed below. They are described in the next section.

**TRANSFORMATION FUNCTIONS**

```
VECTOR_MAP RESIZE(VECTOR_MAP Vector_Map [, VECTOR Size ])
VECTOR_MAP CROP(VECTOR_MAP Vector_Map)
```

## WARP_MAP

**TYPE NAME**

WARP_MAP

**CONSTRUCTOR FUNCTIONS**

```
WARP_MAP WARP_MAP(FILE_NAME File_Name)
```

## DESCRIPTION

A WARP_MAP describes an arbitrary image warp function as a discrete set of mappings from output coordinates to input coordinates, i.e. it describes a two-dimensional array of input x and y coordinates. The warp map is used in image warping (refer to the WARP function).

A WARP_MAP is constructed from a named warp map file. The format of a warp map file is described in Section 10.17.

# 7 Image Processing Functions

## ADD(IMAGE, COLOR)

### SYNOPSIS

```
IMAGE ADD(IMAGE Image, COLOR Color)
```

### PARAMETERS

| | |
|---|---|
| *Image* | Source image |
| *Color* | Color to add |

### DESCRIPTION

Returns a copy of the pixel-component-wise sum of the source image and the color.

Pixel component values are truncated to 1.0.

## ADD(IMAGE, IMAGE)

### SYNOPSIS

```
IMAGE ADD(IMAGE Image1, IMAGE Image2)
```

### PARAMETERS

| | |
|---|---|
| *Image1* | First source image |
| *Image2* | Second source image |

### DESCRIPTION

Returns a copy of the pixel-component-wise sum of the two source images.

Pixel component values are truncated to 1.0.

## ADD_NOISE(IMAGE, real)

### SYNOPSIS

```
IMAGE ADD_NOISE(IMAGE Image, real Amount)
```

### PARAMETERS

| | |
|---|---|
| *Image* | Source image |
| *Amount* | Amount of noise |

### DESCRIPTION

Returns a copy of the source image with uniform noise added.

Independent noise is added to each pixel component value.

The noise has a mean of zero and a range of the specified amount.

## ADD_MONOCHROME_NOISE(IMAGE, real)

### SYNOPSIS

```
IMAGE ADD_MONOCHROME_NOISE(
      IMAGE Image,
      real Amount)
```

### PARAMETERS

| | |
|---|---|
| *Image* | Source image |
| *Amount* | Amount of noise |

**DESCRIPTION**

Returns a copy of the source image with uniform monochrome noise added.

The same noise is added to each pixel component value.

The noise has a mean of zero and a range of the specified amount.

## AREA_THRESHOLD(IMAGE, real)

**SYNOPSIS**

```
IMAGE AREA_THRESHOLD(IMAGE Image, real Proportion)
```

**PARAMETERS**

| | |
|---|---|
| *Image* | Source image |
| *Proportion* | Proportion of area |

**DESCRIPTION**

Returns a thresholded copy of the source image, where the threshold value is automatically chosen so that no more than the specified proportion of pixels are turned on. Refer to the THRESHOLD function for details about thresholding.

## ART_TILE(IMAGE, MATTE, TILE_LAYER_LIST, TILE_OPTIONS)

**SYNOPSIS**

```
IMAGE ART_TILE(
     IMAGE Image,
     MATTE Layer_Selection,
     TILE_LAYER_LIST Layer_List
     [, TILE_OPTIONS Options ])
```

**PARAMETERS**

| | |
|---|---|
| *Image* | Source image |
| *Layer_Selection* | Layer selection |
| *Layer_List* | Layer list |
| *Options* | Optional list of tile options |

**DESCRIPTION**

Returns an artistically tiled rendition of the source image. Artistic tiling is described in Section 2.1. Basic artistic painting via tiling is described in Section **Error! Reference source not found.**.

Each tile layer in the specified tile layer list describes a tile pattern and a layer threshold (refer to the TILE_LAYER and TILE_PATTERN data types).

The tiling of a single layer proceeds as described for the single-pattern ART_TILE function below, with the specified source image, with the tile pattern of the appropriate layer, and with the specified options.

Multi-layer tiling proceeds one layer at a time, starting with the first (or bottom) layer in the layer list, and finishing with the last (or top) layer in the list. Each layer is composited with the previous layers. The first layer is composited with the optional background image (refer to the TILE_OPTION data type).

Within a layer, a tile is only placed if the average opacity of the region which it covers in the specified layer selection matte does not exceed the layer threshold.

## ART_TILE(IMAGE, TILE_PATTERN, TILE_OPTIONS)

**SYNOPSIS**

```
IMAGE ART_TILE(
     IMAGE Image,
     TILE_PATTERN Tile_Pattern
```

```
[, TILE_OPTIONS Options ])
```

**PARAMETERS**

| | |
|---|---|
| *Image* | Source image |
| *Tile_Pattern* | Tile pattern |
| *Options* | Optional list of tile options |

**DESCRIPTION**

Returns an artistically tiled rendition of the source image. Artistic tiling is described in Section 2.1. Basic artistic painting via tiling is described in Section **Error! Reference source not found.**.

The tile pattern describes a collection of tiles and how to place them to fully tile the image (refer to the `TILE_PATTERN` and `TILE` data types).

Tiling starts in the top left corner of the source image. It proceeds by repeatedly placing the tile pattern, spaced appropriately, and placing the tiles within the pattern, spaced appropriately. The options determine the appearance of each tile – the way the tile color is derived from the corresponding region of the source image, and way the surface texture of the tile is combined with the cumulative surface texture of the output.

## AXIS_POINT(CLIP_IMAGE)

**SYNOPSIS**

```
POINT AXIS_POINT(CLIP_IMAGE Clip_Image)
```

**PARAMETERS**

| | |
|---|---|
| *Clip_Image* | Source clip image |

**DESCRIPTION**

Returns the axis point of the source clip image.

## BLUR(IMAGE)

**SYNOPSIS**

```
IMAGE BLUR(IMAGE Image, real Radius)
```

**PARAMETERS**

| | |
|---|---|
| *Image* | Source image |
| *Radius* | Blur radius |

**DESCRIPTION**

Returns a copy of the source image, blurred to the specified radius.

Note that unlike other (pixel-oriented) filter functions, the blur radius specifies model units, not pixels.

## BOTTOM_EDGE()

**SYNOPSIS**

```
real BOTTOM_EDGE()
```

**DESCRIPTION**

Returns the y coordinate of the bottom edge of the working region .

## CHARACTER_ADVANCE(character, FONT)

**SYNOPSIS**

```
real CHARACTER_ADVANCE(character Character, FONT Font)
```

**PARAMETERS**

| | |
|---|---|
| *Character* | Source character |

| | |
|---|---|
| *Font* | Target font |

**DESCRIPTION**

Returns the advance distance for the specified character in the specified font.

## CHARACTER_ADVANCE(character, IMAGE_FONT)

**SYNOPSIS**

```
real CHARACTER_ADVANCE(
     character Character,
     IMAGE_FONT Image_Font)
```

**PARAMETERS**

| | |
|---|---|
| *Character* | Source character |
| *Image_Font* | Target image font |

**DESCRIPTION**

Returns the advance distance for the specified character in the specified image font.

## COLOR_BLEND(COLOR, COLOR, COLOR, COLOR, COLOR_SPACE)

**SYNOPSIS**

```
IMAGE COLOR_BLEND(
     COLOR Color1,
     COLOR Color2,
     COLOR Color3,
     COLOR Color4
     [, COLOR_SPACE Color_Space ])
```

**PARAMETERS**

| | |
|---|---|
| *Color1* | Top left color |
| *Color2* | Top right color |
| *Color3* | Bottom right color |
| *Color4* | Bottom left color |
| *Color_Space* | Optional target color space; default value is the working color space |

**DESCRIPTION**

Returns a new image of the working size containing a four-point color blend, in the specified color space.

## COMPOSITE(CLIP_IMAGE, IMAGE, POINT, real)

**SYNOPSIS**

```
IMAGE COMPOSITE(
     CLIP_IMAGE Clip_Image,
     IMAGE Background_Image,
     POINT Position
     [, real Height ])
COMPOSITE(
     CLIP_IMAGE Clip_Image,
     variable IMAGE Background_Image,
     POINT Position
     [, real Height ])
```

**PARAMETERS**

| | |
|---|---|
| *Clip_Image* | Foreground clip image |
| *Background_Image* | Background image |
| *Position* | Target position |
| *Height* | Optional height above the image plane; default value is 0.0 |

**DESCRIPTION**

Returns a copy of the foreground clip image composited over the background image. Modifies the background image in the procedure version.

Each foreground pixel value, already pre-multiplied by its associated opacity value, is interpolated with the corresponding background pixel value according to its opacity value:

$$foreground + background \times (1.0 - opacity)$$

The optional height takes effect when Vark is running in stereoscopic mode.

## COMPOSITE(IMAGE, MATTE, IMAGE)

**SYNOPSIS**

```
IMAGE COMPOSITE(
      IMAGE Image,
      IMAGE Matte,
      IMAGE Background_Image
      [, real Height ])
```

**PARAMETERS**

| | |
|---|---|
| *Image* | Foreground image |
| *Matte* | Matte |
| *Background_Image* | Background image |
| *Height* | Optional height above the image plane; default value is 0.0 |

**DESCRIPTION**

Returns a copy of the foreground image composited over the background image.

Each foreground pixel value is interpolated with the corresponding background pixel value according to the corresponding opacity value from the matte:

$$background + (foreground - background) \times opacity$$

The foreground image and matte are cropped to the size of the background image if necessary.

The optional height takes effect when Vark is running in stereoscopic mode.

## CONVERT_COLOR_SPACE(IMAGE, COLOR_SPACE)

**SYNOPSIS**

```
IMAGE CONVERT_COLOR_SPACE(
      IMAGE Image,
      COLOR_SPACE Color_Space)
```

**PARAMETERS**

| | |
|---|---|
| *Image* | Source image |
| *Color_Space* | Target color space |

**DESCRIPTION**

Returns a copy of the source image converted to the target color space.

## CONVOLVE(IMAGE, KERNEL, boolean)

**SYNOPSIS**

```
IMAGE CONVOLVE(
      IMAGE Image,
      KERNEL Kernel
      [, boolean Absolute_Value ])
```

**PARAMETERS**

| | |
|---|---|
| *Image* | Source image |
| *Kernel* | Convolution kernel |
| *Absolute_Value* | Optional absolute value flag; default value is `false` |

**DESCRIPTION**

Returns a copy of the convolution of the source image with the specified convolution kernel.

The optional *absolute value* flag specifies, if `true`, that the absolute value of negative intermediate pixel values be taken. If the flag is `false`, then negative pixel values are truncated.

## CROP(BUMP_MAP)

**SYNOPSIS**

BUMP_MAP **CROP**(BUMP_MAP *Bump_Map*)

**PARAMETERS**

| | |
|---|---|
| *Bump_Map* | Source bump map |

**DESCRIPTION**

Returns a copy of the source bump map cropped to the working size.

The returned bump map has a value of zero where the source bump map is undefined, i.e. in the case where the source bump map is smaller than the working size.

## CROP(CLIP_IMAGE)

**SYNOPSIS**

CLIP_IMAGE **CROP**(CLIP_IMAGE *Clip_Image*)

**PARAMETERS**

| | |
|---|---|
| *Clip_Image* | Source clip image |

**DESCRIPTION**

Returns a copy of the source clip image cropped to the working size.

The returned clip image has a value of zero where the source clip image is undefined, i.e. in the case where the source clip image is smaller than the working size.

## CROP(DISPLACEMENT_MAP)

**SYNOPSIS**

DISPLACEMENT_MAP **CROP**(DISPLACEMENT_MAP *Displacement_Map*)

**PARAMETERS**

| | |
|---|---|
| *Displacement _Map* | Source displacement map |

**DESCRIPTION**

Returns a copy of the source displacement map cropped to the working size.

The returned displacement map has a value of zero where the source displacement map is undefined, i.e. in the case where the source displacement map is smaller than the working size.

## CROP(IMAGE)

**SYNOPSIS**

IMAGE **CROP**(IMAGE *Image*)

**PARAMETERS**

| | |
|---|---|
| *Image* | Source image |

**DESCRIPTION**

Returns a copy of the source image cropped to the working size.

The returned image has a value of zero where the source image is undefined, i.e. in the case where the source image is smaller than the working size.

## CROP(IMAGE, real, real)

**SYNOPSIS**

```
IMAGE CROP(IMAGE Image, real X_Size, real Y_Size)
```

**PARAMETERS**

| | |
|---|---|
| *Image* | Source image |
| *X_Size* | Crop size in X dimension |
| *Y_Size* | Crop size in Y dimension |

**DESCRIPTION**

Returns a copy of the source image cropped to the specified size, centered.

The image is black where the source image is undefined, i.e. in the case where the source image is smaller than the specified size.

## CROP(IMAGE, RECTANGLE)

**SYNOPSIS**

```
IMAGE CROP(IMAGE Image, RECTANGLE Region)
```

**PARAMETERS**

| | |
|---|---|
| *Image* | Source image |
| *Region* | Crop region |

**DESCRIPTION**

Returns a copy of the source image cropped to the specified region.

The image is black where the source image is undefined, i.e. in the case where the source image only partially covers the crop region.

## CROP(MATTE)

**SYNOPSIS**

```
MATTE CROP(MATTE Matte)
```

**PARAMETERS**

| | |
|---|---|
| *Matte* | Source matte |

**DESCRIPTION**

Returns a copy of the source matte cropped to the working size.

The returned matte has a value of zero where the source matte is undefined, i.e. in the case where the source matte is smaller than the working size.

## DESATURATE(IMAGE)

**SYNOPSIS**

```
IMAGE DESATURATE(IMAGE Image)
```

**PARAMETERS**

| | |
|---|---|
| *Image* | Source image |

**DESCRIPTION**

Returns a desaturated copy of the source image.

Each pixel value is desaturated by replacing it with its luminance value.

## DETECT_DETAILED_FACE(IMAGE)

**SYNOPSIS**

```
FACE DETECT_DETAILED_FACE(IMAGE Image)
```

**PARAMETERS**

| | |
|---|---|
| *Image* | Source image |

**DESCRIPTION**

Returns a FACE which describes the face with the highest face metric in the source image.

## DETECT_DETAILED_FACES(IMAGE, integer, integer, real)

**SYNOPSIS**

```
FACE_LIST DETECT_DETAILED_FACES(
     IMAGE Image
     [, integer Maximum_Faces
     [, real Face_Threshold ]])
```

**PARAMETERS**

| | |
|---|---|
| *Image* | Source image |
| *Maximum_Faces* | Optional maximum number of faces to detect; default value is unlimited |
| *Face_Threshold* | Optional face metric threshold; default value is 0.9 |

**DESCRIPTION**

Returns a FACE_LIST, each FACE of which identifies a face in the source image. Faces are listed in order of the strength of their face metric.

## DETECT_FACE(IMAGE)

**SYNOPSIS**

```
RECTANGLE DETECT_FACE(IMAGE Image)
```

**PARAMETERS**

| | |
|---|---|
| *Image* | Source image |

**DESCRIPTION**

Returns a rectangle which identifies the face with the highest face metric in the source image. The rectangle corresponds roughly to the Head_Region member of the corresponding FACE.

## DETECT_FACES(IMAGE, integer, integer, real)

**SYNOPSIS**

```
RECTANGLE_LIST DETECT_FACES(
     IMAGE Image
     [, integer Maximum_Faces
     [, real Face_Threshold ]])
```

**PARAMETERS**

| | |
|---|---|
| *Image* | Source image |
| *Maximum_Faces* | Optional maximum number of faces to detect; |

| | |
|---|---|
| | default value is unlimited |
| *Face_Threshold* | Optional face metric threshold; default value is 0.9 |

**DESCRIPTION**

Returns a list of rectangles, each of which identifies a human face in the source image. Faces are listed in order of the strength of their face metric. Each rectangle corresponds roughly to the `Head_Region` member of the corresponding FACE.

## DITHER(IMAGE, DITHER_MATRIX, COLOR_LIST)

**SYNOPSIS**

```
IMAGE DITHER(
     IMAGE Image,
     DITHER_MATRIX Dither_Matrix
     [, COLOR_LIST Color_List ])
```

**PARAMETERS**

| | |
|---|---|
| *Image* | Source image |
| *Dither_Matrix* | Target dither matrix |
| *Color_List* | Optional target color list |

**DESCRIPTION**

Returns a dithered copy of the source image.

Each pixel component value is dithered by replacing it with 1.0 if it is greater than or equal to the corresponding value in the dither matrix, and replacing it with 0.0 otherwise.

The row and column of a pixel in the source image addresses a corresponding location in the dither matrix, modulo the pixel height and width of the dither matrix.

If the optional color list is specified, then each pixel component value is dithered between the nearest corresponding channel values in the color list, rather than between 0.0 and 1.0.

## EXTRACT_CHANNEL(IMAGE, CHANNEL)

**SYNOPSIS**

```
IMAGE EXTRACT_CHANNEL(
     IMAGE Image,
     CHANNEL Channel)
```

**PARAMETERS**

| | |
|---|---|
| *Image* | Source image |
| *Channel* | Channel to extract |

**DESCRIPTION**

Returns a copy of the specified channel in the source image. The extracted image is a single-channel image of custom color space.

The channel specification must be consistent with the color space of the source image.

## FIND_EDGES(IMAGE)

**SYNOPSIS**

```
IMAGE FIND_EDGES(IMAGE Image)
```

**PARAMETERS**

| | |
|---|---|
| *Image* | Source image |

**DESCRIPTION**

Returns a new RGB image whose pixel values reflect pixel-component-wise edge strength in the source image.

Edge strength is computed as the maximum of a horizontal derivative filter and a vertical derivative filter of the source image.

## FLIP_HORIZONTALLY(IMAGE)

**SYNOPSIS**

```
IMAGE FLIP_HORIZONTALLY(IMAGE Image)
```

**PARAMETERS**

| | |
|---|---|
| *Image* | Source image |

**DESCRIPTION**

Returns a copy of the source image flipped horizontally.

## FLIP_VERTICALLY(IMAGE)

**SYNOPSIS**

```
IMAGE FLIP_VERTICALLY(IMAGE Image)
```

**PARAMETERS**

| | |
|---|---|
| *Image* | Source image |

**DESCRIPTION**

Returns a copy of the source image flipped vertically.

## HORIZONTAL_COLOR_BLEND(COLOR, COLOR, COLOR_SPACE)

**SYNOPSIS**

```
IMAGE HORIZONTAL_COLOR_BLEND(
      COLOR Start_Color,
      COLOR End_Color
      [, COLOR_SPACE Color_Space ])
```

**PARAMETERS**

| | |
|---|---|
| *Start_Color* | Blend start color (left) |
| *End_Color* | Blend end color (right) |
| *Color_Space* | Optional target color space; default value is the working color space |

**DESCRIPTION**

Returns a new image of the working size containing a horizontal color blend from the start color on the left to the end color on the right, in the specified color space.

## ILLUMINATE(IMAGE, ILLUMINATION_OPTIONS, LIGHT_LIST, BUMP_MAP)

**SYNOPSIS**

```
IMAGE ILLUMINATE(
      IMAGE Image,
      ILLUMINATION_OPTIONS Options,
      LIGHT_LIST Light_List
      [, BUMP_MAP Bump_Map ])
```

**PARAMETERS**

| | |
|---|---|
| *Image* | Source image |
| *Options* | List of illumination options |
| *Light_List* | List of lights |
| *Bump_Map* | Optional bump map |

## DESCRIPTION

Returns an illuminated copy of the source image.

If a bump map is specified or the source image has a bump map channel, then the lighting uses the bump map to determine the surface texture of the image for lighting purposes. In the absence of a bump map the surface is taken to be flat.

The illumination options specify optional surface characteristics, including ambient, diffuse and specular reflection characteristics, and shadow depth – i.e. "bumpiness" (refer to the `ILLUMINATION_OPTION` data type).

The light list specifies one or more light sources (refer to the `LIGHT` data type).

The lighting of a pixel in the source image is computed as the sum of the contribution from the ambient light and the contribution from each light source, according to the equations given below.

The intensity of a light source at a finite distance is attenuated with the square of the distance, but the attenuation factor is normalised to 1.0 at the closest point in the image to the light source. Thus moving the light source closer to the image leaves the intensity at the closest point unchanged, but progressively darkens the rest of the image. And moving the light source further away progressively lightens the rest of the image, until the light source intensity is constant across the image when the light source is at infinity.

## EQUATIONS

The normal surface vector ($N$) at a pixel is computed from the bump map, if present. The default normal vector in the absence of a bump map is perpendicular to the image plane.

The viewing vector ($V$) is always perpendicular to the image plane.

For a light source at infinity, the light source vector ($L$) from a pixel to the light source is constant across the entire image, so is computed once for the entire image. For a light source at a finite distance, the light source vector is computed independently for each pixel.

A pixel's reflection of ambient light is computed according to:

$$I_a k_a O_d$$

A pixel's diffuse and specular reflection of a light source is computed according to the Phong model:

$$f_{att} I_p [\, k_d O_d (N \bullet L) + k_s O_s (N \bullet H)^n \,]$$

| | |
|---|---|
| $d_L$ | distance from light source |
| $f_{att}$ | attenuation with distance ($f_{att} = 1 / d_L^2$) |
| $H$ | normalised half-way vector between $L$ and $V$ |
| $I_a$ | ambient light intensity |
| $I_p$ | point light source intensity |
| $k_a$ | ambient reflection coefficient |
| $k_d$ | diffuse reflection coefficient |
| $k_s$ | specular reflection coefficient |
| $k_{sc}$ | specular color coefficient |
| $L$ | normalised light source vector |
| $N$ | normalised surface normal vector |
| $n$ | specular exponent |
| $O_d$ | object's diffuse color (i.e. image pixel color) |
| $O_s$ | object's specular color ($k_{sc} O_d + (1 - k_{sc}) I_p$) |
| $V$ | normalised viewing vector |

The same reflection coefficients ($k_a$, $k_s$, $k_d$) are used for each color component.

### INVERT(IMAGE)

**SYNOPSIS**

```
IMAGE INVERT(IMAGE Image)
```

**PARAMETERS**

| | |
|---|---|
| *Image* | Source image |

**DESCRIPTION**

Returns a copy of the source image with its colors inverted.

Each pixel component value is inverted by subtracting it from 1.0.

### LEFT_EDGE()

**SYNOPSIS**

```
real LEFT_EDGE()
```

**DESCRIPTION**

Returns the x coordinate of the left edge of the working region .

### LOOKUP_COLOR(IMAGE, CLUT)

**SYNOPSIS**

```
IMAGE LOOKUP_COLOR(IMAGE Image, CLUT Table)
```

**PARAMETERS**

| | |
|---|---|
| *Image* | Source image |
| *Table* | Color lookup table (CLUT) |

**DESCRIPTION**

Returns a copy of the source image with its colors translated via the specified color lookup table.

Each image channel is translated via a separate channel in the color lookup table. Each pixel component value is translated by replacing it with the value it indexes in the table.

### LOOKUP_COLOR(IMAGE, CLUT_3D)

**SYNOPSIS**

```
IMAGE LOOKUP_COLOR(IMAGE Image, CLUT_3D Table)
```

**PARAMETERS**

| | |
|---|---|
| *Image* | Source image |
| *Table* | 3D color lookup table (3D CLUT) |

**DESCRIPTION**

Returns a copy of the source image with its colors translated via the specified 3D color lookup table.

Each image color is translated by using its three color components to index into the 3D color lookup table. Because the 3D color lookup table is sparse, the eight colors adjacent to the indexed location are interpolated to yield the translated color.

### MAX(IMAGE, IMAGE)

**SYNOPSIS**

```
IMAGE MAX(IMAGE Image1, IMAGE Image2)
```

**PARAMETERS**

| | |
|---|---|
| *Image1* | First source image |
| *Image2* | Second source image |

**DESCRIPTION**

Returns a copy of the pixel-component-wise maximum of the two source images.

The two source images must have the same size and color space.

## MAX_FILTER(IMAGE, integer)

**SYNOPSIS**

```
IMAGE MAX_FILTER(
     IMAGE Image
     [, integer Radius ])
```

**PARAMETERS**

| | |
|---|---|
| *Image* | Source image |
| *Radius* | Filter radius, in pixels; default 1 |

**DESCRIPTION**

Returns a maximum filtered copy of the source image.

The filter is computed by replacing each pixel component value by the maximum value in its square neighborhood. The radius specifies the radius of the neighborhood.

## MEAN_MIN_MAX_FILTER(IMAGE, integer)

**SYNOPSIS**

```
IMAGE MEAN_MIN_MAX_FILTER(
     IMAGE Image
     [, integer Radius ])
```

**PARAMETERS**

| | |
|---|---|
| *Image* | Source image |
| *Radius* | Filter radius, in pixels; default 1 |

**DESCRIPTION**

Returns a mean of minimum and maximum filtered copy of the source image.

The filter is computed by replacing each pixel component value by the mean of the minimum and maximum values in its square neighborhood. The radius specifies the radius of the neighborhood.

## MEDIAN_FILTER(IMAGE, integer)

**SYNOPSIS**

```
IMAGE MEDIAN_FILTER(
     IMAGE Image
     [, integer Radius ])
```

**PARAMETERS**

| | |
|---|---|
| *Image* | Source image |
| *Radius* | Filter radius, in pixels; default 1 |

**DESCRIPTION**

Returns a median filtered copy of the source image.

The filter is computed by replacing each pixel component value by the median value in its square neighborhood. The radius specifies the radius of the neighborhood.

## MIN(IMAGE, IMAGE)

**SYNOPSIS**

```
IMAGE MIN(IMAGE Image1, IMAGE Image2)
```

**PARAMETERS**

| | |
|---|---|
| *Image1* | First source image |
| *Image2* | Second source image |

**DESCRIPTION**

Returns a copy of the pixel-component-wise minimum of the two source images.

The two source images must have the same size and color space.

## MIN_FILTER(IMAGE, integer)

**SYNOPSIS**

```
IMAGE MIN_FILTER(
     IMAGE Image
     [, integer Radius ])
```

**PARAMETERS**

| | |
|---|---|
| *Image* | Source image |
| *Radius* | Filter radius, in pixels; default 1 |

**DESCRIPTION**

Returns a minimum filtered copy of the source image.

The filter is computed by replacing each pixel component value by the minimum value in its square neighborhood. The radius specifies the radius of the neighborhood.

## MULTIPLY(IMAGE, COLOR)

**SYNOPSIS**

```
IMAGE MULTIPLY(IMAGE Image, COLOR Color)
```

**PARAMETERS**

| | |
|---|---|
| *Image* | Source image |
| *Color* | Color to muliply by |

**DESCRIPTION**

Returns a copy of the pixel-component-wise product of the source image and the color.

## MULTIPLY(IMAGE, IMAGE)

**SYNOPSIS**

```
IMAGE MULTIPLY(IMAGE Image1, IMAGE Image2)
```

**PARAMETERS**

| | |
|---|---|
| *Image1* | First source image |
| *Image2* | Second source image |

**DESCRIPTION**

Returns a copy of the pixel-component-wise product of the two source images.

## MULTIPLY(MATTE, real)

**SYNOPSIS**

```
MATTE MULTIPLY(MATTE Matte, real Opacity)
```

**PARAMETERS**

| | |
|---|---|
| *Matte* | Source matte |
| *Opacity* | Opacity to muliply by |

**DESCRIPTION**

Returns a copy of the pixel-component-wise product of the source matte and the opacity.

## PHOTO()

### SYNOPSIS

```
IMAGE PHOTO()
```

### DESCRIPTION

Returns a copy of the input photo image. The input photo image is enhanced in various ways specific to the input device.

## POSTERIZE(IMAGE, integer)

### SYNOPSIS

```
IMAGE POSTERIZE(IMAGE Image, integer Levels)
```

### PARAMETERS

| | |
|---|---|
| *Image* | Source image |
| *Levels* | Number of target levels |

### DESCRIPTION

Returns a posterized copy of the source image.

The number of target levels specifies the number of equally-spaced target color component values in the range 0.0 to 1.0. The minimum number of target levels is 2.

Each pixel component value is posterized by replacing it by its nearest level value.

## PRINT(IMAGE)

### SYNOPSIS

```
PRINT(IMAGE Image)
```

### PARAMETERS

| | |
|---|---|
| *Image* | Source image |

### DESCRIPTION

Prints, or otherwise outputs, the source image to the current output device.

## RESIZE(BUMP_MAP, VECTOR)

### SYNOPSIS

```
BUMP_MAP RESIZE(BUMP_MAP Bump_Map [, VECTOR Size ])
```

### PARAMETERS

| | |
|---|---|
| *Bump_Map* | Source bump map |
| *Size* | Optional target size; default value is working size |

### DESCRIPTION

Returns a copy of the source bump map scaled to the specified size, or to the working size if no size is specified.

## RESIZE(CLIP_IMAGE, VECTOR)

### SYNOPSIS

```
CLIP_IMAGE RESIZE(CLIP_IMAGE Clip_Image [, VECTOR Size ])
```

### PARAMETERS

| | |
|---|---|
| *Clip_Image* | Source clip image |
| *Size* | Optional target size; default value is working size |

### DESCRIPTION

Returns a copy of the source clip image scaled to the specified size, or to the working size if no size is specified.

## RESIZE(DISPLACEMENT_MAP, VECTOR)

### SYNOPSIS

```
DISPLACEMENT_MAP RESIZE(
    DISPLACEMENT_MAP Displacement_Map
    [, VECTOR Size ])
```

### PARAMETERS

| | |
|---|---|
| *Displacement_Map* | Source displacement map |
| *Size* | Optional target size; default value is working size |

### DESCRIPTION

Returns a copy of the source displacement map scaled to the specified size, or to the working size if no size is specified.

## RESIZE(IMAGE, VECTOR)

### SYNOPSIS

```
IMAGE RESIZE(IMAGE Image [, VECTOR Size ])
```

### PARAMETERS

| | |
|---|---|
| *Image* | Source image |
| *Size* | Optional target size; default value is working size |

### DESCRIPTION

Returns a copy of the source image scaled to the specified size, or to the working size if no size is specified.

## RESIZE(MATTE, VECTOR)

### SYNOPSIS

```
MATTE RESIZE(MATTE Matte [, VECTOR Size ])
```

### PARAMETERS

| | |
|---|---|
| *Matte* | Source matte |
| *Size* | Optional target size; default value is working size |

### DESCRIPTION

Returns a copy of the source matte scaled to the specified size, or to the working size if no size is specified.

## RIGHT_EDGE()

### SYNOPSIS

```
real RIGHT_EDGE()
```

### DESCRIPTION

Returns the x coordinate of the right edge of the working region .

## QUANTIZE(IMAGE, PALETTE)

### SYNOPSIS

```
IMAGE QUANTIZE(IMAGE Image, PALETTE Palette)
```

### PARAMETERS

| | |
|---|---|
| *Image* | Source image |
| *Palette* | Target palette |

### DESCRIPTION

Returns a copy of the source image with its colors quantized.

Each pixel value is quantized by replacing it with the nearest color in the target palette.

## QUANTIZE_HUE(IMAGE, integer)

**SYNOPSIS**

```
IMAGE QUANTIZE_HUE(IMAGE Image, integer Levels)
```

**PARAMETERS**

| | |
|---|---|
| *Image* | Source image |
| *Levels* | Number of target levels |

**DESCRIPTION**

Returns a copy of the source image with its hue values quantized to the number of target levels.

QUANTIZE_HUE builds a hue palette with the required number of target levels by finding the highest peaks in the hue histogram of the source image. Each pixel's hue component value is quantized by replacing it with the nearest hue in the hue palette.

The color space of the source image must be HSV.

## RAW_PHOTO()

**SYNOPSIS**

```
IMAGE RAW_PHOTO()
```

**DESCRIPTION**

Returns a copy of the raw input photo image. The raw input photo image is not enhanced in any way.

## RENDER_TEXT(TEXT, FONT, COLOR, TEXT_OPTIONS)

**SYNOPSIS**

```
CLIP_IMAGE RENDER_TEXT(
     TEXT Text,
     FONT Font,
     COLOR Color
     [, TEXT_OPTIONS Options ])
```

**PARAMETERS**

| | |
|---|---|
| *Text* | Text to set |
| *Font* | Font to set text in |
| *Color* | Text color |
| *Options* | Optional list of text setting options |

**DESCRIPTION**

Returns a clip image containing the text set and rendered in the specified outline font, according to the specified text setting options (refer to the TEXT_OPTION data type), in the specified color.

## RENDER_TEXT(TEXT, IMAGE_FONT, TEXT_OPTIONS)

**SYNOPSIS**

```
CLIP_IMAGE RENDER_TEXT(
     TEXT Text,
     IMAGE_FONT Image_Font
     [, TEXT_OPTIONS Options ])
```

**PARAMETERS**

| | |
|---|---|
| *Text* | Text to set |
| *Image_Font* | Image font to set text in |

| | |
|---|---|
| *Options* | Optional list of text setting options |

**DESCRIPTION**

Returns a clip image containing the text set and rendered in the specified image font, according to the specified text setting options (refer to the TEXT_OPTION data type).

## RENDER_TEXT(TEXT, IMAGE_FONT, TEXT_OPTIONS)

**SYNOPSIS**

```
CLIP_IMAGE RENDER_TEXT(
     TEXT Text,
     IMAGE_FONT_3D Image_Font_3D
     [, TEXT_OPTIONS Options ])
```

**PARAMETERS**

| | |
|---|---|
| *Text* | Text to set |
| *Image_Font_3D* | 3D image font to set text in |
| *Options* | Optional list of text setting options |

**DESCRIPTION**

Returns a clip image containing the text set and rendered in the specified 3D image font, according to the specified text setting options (refer to the TEXT_OPTION data type).

## REPLACE_CHANNEL(IMAGE, CHANNEL, IMAGE)

**SYNOPSIS**

```
IMAGE REPLACE_CHANNEL(
     IMAGE Image,
     CHANNEL Channel,
     IMAGE Channel_Image)
```

**PARAMETERS**

| | |
|---|---|
| *Image* | Source image |
| *Channel* | Channel to replace |
| *Channel_Image* | Image containing replacement channel |

**DESCRIPTION**

Returns a new image containing a version of the source image with the specified channel replaced with the channel from the channel image.

The channel specification must be consistent with the color space of the source image.

The channel image must be a single-channel image.

## REPLACE_CHANNEL(IMAGE, CHANNEL, real)

**SYNOPSIS**

```
IMAGE REPLACE_CHANNEL(
     IMAGE Image,
     CHANNEL Channel,
     real Value)
```

**PARAMETERS**

| | |
|---|---|
| *Image* | Source image |
| *Channel* | Channel to replace |
| *Value* | Replacement value |

**DESCRIPTION**

Returns a new image containing a version of the source image with the specified channel replaced with the specified replacement value.

The channel specification must be consistent with the color space of the source image.

## ROTATE(CLIP_IMAGE, real)

**SYNOPSIS**

```
CLIP_IMAGE ROTATE(
    CLIP_IMAGE Clip_Image,
    real Angle)
```

**PARAMETERS**

| | |
|---|---|
| *Clip_Image* | Source clip image |
| *Angle* | Rotation angle, in degrees |

**DESCRIPTION**

Returns a copy of the source clip image rotated counter-clockwise by the specified angle.

## ROTATE(IMAGE, real)

**SYNOPSIS**

```
IMAGE ROTATE(IMAGE Image, real Angle)
```

**PARAMETERS**

| | |
|---|---|
| *Image* | Source image |
| *Angle* | Rotation angle, in degrees |

**DESCRIPTION**

Returns a copy of the source image rotated counter-clockwise by the specified angle.

The image is black where the source image is undefined.

## SAVE(CLUT_3D, FILE_NAME)

**SYNOPSIS**

```
SAVE(CLUT_3D Table, FILE_NAME File_Name)
```

**PARAMETERS**

| | |
|---|---|
| *Table* | Source 3D CLUT |
| *File_Name* | Name of target file |

**DESCRIPTION**

Saves the source 3D CLUT to a file with the specified name.

This procedure is intended for diagnostic use in a non-embedded environment.

## SAVE(IMAGE, FILE_NAME)

**SYNOPSIS**

```
SAVE(IMAGE Image, FILE_NAME File_Name)
```

**PARAMETERS**

| | |
|---|---|
| *Image* | Source image |
| *File_Name* | Name of target file |

**DESCRIPTION**

Saves the source image to a file with the specified name.

This procedure is intended for diagnostic use in a non-embedded environment.

## SAVE(KERNEL, FILE_NAME)

**SYNOPSIS**

```
SAVE(KERNEL Kernel, FILE_NAME File_Name)
```

**PARAMETERS**

| | |
|---|---|
| *Kernel* | Source convolution kernel |
| *File_Name* | Name of target file |

**DESCRIPTION**

Saves the source convolution kernel to a file with the specified name.

This procedure is intended for diagnostic use in a non-embedded environment.

## SAVE(PALETTE, FILE_NAME)

**SYNOPSIS**

```
SAVE(PALETTE Palette, FILE_NAME File_Name)
```

**PARAMETERS**

| | |
|---|---|
| *Palette* | Source palette |
| *File_Name* | Name of target file |

**DESCRIPTION**

Saves the source palette to a file with the specified name.

This procedure is intended for diagnostic use in a non-embedded environment.

## SCALE(CLIP_IMAGE, real)

**SYNOPSIS**

```
CLIP_IMAGE SCALE(
     CLIP_IMAGE Clip_Image,
     real Scale)
```

**PARAMETERS**

| | |
|---|---|
| *Clip_Image* | Source clip image |
| *Scale* | Scale factor |

**DESCRIPTION**

Returns a copy of the source clip image scaled by the specified factor.

## SCALE(CLIP_IMAGE, real, real)

**SYNOPSIS**

```
CLIP_IMAGE SCALE(
     CLIP_IMAGE Clip_Image,
     real X_Scale,
     real Y_Scale)
```

**PARAMETERS**

| | |
|---|---|
| *Clip_Image* | Source clip image |
| *X_Scale* | Scale factor in X dimension |
| *Y_Scale* | Scale factor in Y dimension |

**DESCRIPTION**

Returns a copy of the source clip image scaled by the specified factors.

## SCALE(IMAGE, real)

**SYNOPSIS**

```
IMAGE SCALE(IMAGE Image, real Scale)
```

**PARAMETERS**

| | |
|---|---|
| *Image* | Source image |
| *Scale* | Scale factor |

**DESCRIPTION**

Returns a copy of the source image scaled by the specified factor.

The scale factor must be positive. Use the FLIP functions to flip images.

## SCALE(IMAGE, real, real)

**SYNOPSIS**

```
IMAGE SCALE(
      IMAGE Image,
      real X_Scale,
      real Y_Scale)
```

**PARAMETERS**

| | |
|---|---|
| *Image* | Source image |
| *X_Scale* | Scale factor in X dimension |
| *Y_Scale* | Scale factor in Y dimension |

**DESCRIPTION**

Returns a copy of the source image scaled by the specified factors.

The scale factors must be positive. Use the FLIP functions to flip images.

## SHADOW(CLIP_IMAGE, real, real, COLOR)

**SYNOPSIS**

```
CLIP_IMAGE SHADOW(
     CLIP_IMAGE Clip_Image,
     real Opacity
     [, real Soft_Edge_Width ]
     [, COLOR Color ])
```

**PARAMETERS**

| | |
|---|---|
| *Clip_Image* | Source clip image |
| *Opacity* | Target opacity |
| *Soft_Edge_Width* | Optional width of soft edge; default value is 0.05 |
| *Color* | Optional shadow color; default value is black |

**DESCRIPTION**

Returns a copy of the source clip image with the color channels set to the specified color, the matte channel scaled by the specified opacity, and the edges of the matte softened to the specified width.

## SIZE(CLIP_IMAGE)

**SYNOPSIS**

```
VECTOR SIZE(CLIP_IMAGE Clip_Image)
```

**PARAMETERS**

| | |
|---|---|
| *Clip_Image* | Source clip image |

**DESCRIPTION**

Returns the size of the source clip image, as a vector.

## SIZE(IMAGE)

**SYNOPSIS**

```
VECTOR SIZE(IMAGE Image)
```

**PARAMETERS**

| | |
|---|---|
| *Image* | Source image |

**DESCRIPTION**

Returns the size of the source image, as a vector.

## SKELETONIZE(IMAGE)

**SYNOPSIS**

```
IMAGE SKELETONIZE(IMAGE Image [, real Threshold ])
```

**PARAMETERS**

| | |
|---|---|
| *Image* | Source image |
| *Threshold* | Optional threshold; default value is 0.5 |

**DESCRIPTION**

Returns a thresholded and skeletonized copy of the source image.

## SKELETONIZE_TO_PATH(IMAGE)

**SYNOPSIS**

```
PATH SKELETONIZE_TO_PATH(
     IMAGE Image
     [, real Threshold ]
     [, real Minimum_Path_Length ])
```

**PARAMETERS**

| | |
|---|---|
| *Image* | Source image |
| *Threshold* | Optional threshold; default value is 0.5 |
| *Minimum_Path_Length* | Optional minimum path length; default value is 0.0 |

**DESCRIPTION**

Thresholds and skeletonizes the source image, and returns a path fitted to the skeleton. Discards sub-paths shorter than the specified minimum length.

## STROKE(PATH, BRUSH, COLOR, IMAGE)

**SYNOPSIS**

```
STROKE(
     PATH Path,
     BRUSH Brush,
     COLOR Color,
     variable IMAGE Image)
```

**PARAMETERS**

| | |
|---|---|
| *Path* | Source path |
| *Brush* | Brush |
| *Color* | Paint color |
| *Image* | Background image |

**DESCRIPTION**

Builds brush strokes along each sub-path of the path and composites the strokes with the background image. Uses a paint of the specified color but with default characteristics. Refer to the **STROKE** function below for more details.

## STROKE(PATH, BRUSH, PAINT, IMAGE)

**SYNOPSIS**

```
STROKE(
     PATH Path,
     BRUSH Brush,
     PAINT Paint,
     variable IMAGE Image)
```

**PARAMETERS**

| | |
|---|---|
| *Path* | Source path |
| *Brush* | Brush |
| *Paint* | Paint |
| *Image* | Background image |

**DESCRIPTION**

Builds brush strokes along each sub-path of the path and composites the strokes with the background image.

Builds a stroke with a start-cap and an end-cap if the corresponding sub-path is open. Builds multiple strokes for a sub-path if the sub-path is longer than the maximum stroke length of the brush. Constrains the rotation of the brush stamp to the allowed rotation range of the brush.

Composites the stroke bump map with the background image bump map according to the paint characteristics, if the background image contains a bump map channel.

## SUBTRACT(IMAGE, COLOR)

**SYNOPSIS**

IMAGE **SUBTRACT**(IMAGE *Image*, COLOR *Color*)

**PARAMETERS**

| | |
|---|---|
| *Image* | Source image |
| *Color* | Color to subtract |

**DESCRIPTION**

Returns a new image containing the pixel-component-wise difference between the source image and the color.

Negative pixel component values are truncated to 0.0.

## SUBTRACT(IMAGE, IMAGE)

**SYNOPSIS**

IMAGE **SUBTRACT**(IMAGE *Image1*, IMAGE *Image2*)

**PARAMETERS**

| | |
|---|---|
| *Image1* | First source image |
| *Image2* | Second source image |

**DESCRIPTION**

Returns a copy of the pixel-component-wise difference between the two source images.

Negative pixel component values are truncated to 0.0.

The two source images must have the same size and color space.

## TESSELATE(BUMP_MAP)

**SYNOPSIS**

BUMP_MAP **TESSELATE**(BUMP_MAP *Bump_Map*)

**PARAMETERS**

| | |
|---|---|
| *Bump_Map* | Source bump map |

**DESCRIPTION**

Returns a new bump map of working size tesselated with copies of the source bump map.

## TESSELATE(IMAGE)

### SYNOPSIS

```
IMAGE TESSELATE(IMAGE Image)
```

### PARAMETERS

| | |
|---|---|
| *Image* | Source image |

### DESCRIPTION

Returns a new image of working size tesselated with copies of the source image.

## THRESHOLD(IMAGE, real)

### SYNOPSIS

```
IMAGE THRESHOLD(IMAGE Image, real Threshold)
```

### PARAMETERS

| | |
|---|---|
| *Image* | Source image |
| *Threshold* | Threshold |

### DESCRIPTION

Returns a thresholded copy of the source image.

Each pixel component value is thresholded by replacing it with 1.0 if it is equal to or greater than the threshold value, and replacing it with 0.0 otherwise.

## TOP_EDGE()

### SYNOPSIS

```
real TOP_EDGE()
```

### DESCRIPTION

Returns the y coordinate of the top edge of the working region .

## TRANSLATE(IMAGE, real, real)

### SYNOPSIS

```
IMAGE TRANSLATE(
     IMAGE Image,
     real X_Offset,
     real Y_Offset)
```

### PARAMETERS

| | |
|---|---|
| *Image* | Source image |
| *X_Offset* | Offset in X dimension |
| *Y_Offset* | Offset in Y dimension |

### DESCRIPTION

Returns a copy of the source image translated by the specified offsets.

The image is black where the source image is undefined.

## VERTICAL_COLOR_BLEND(COLOR, COLOR, COLOR_SPACE)

### SYNOPSIS

```
IMAGE VERTICAL_COLOR_BLEND(
     COLOR Start_Color,
     COLOR End_Color
     [, COLOR_SPACE Color_Space ])
```

**PARAMETERS**

| | |
|---|---|
| *Start_Color* | Blend start color (top) |
| *End_Color* | Blend end color (bottom) |
| *Color_Space* | Optional target color space; default value is the working color space |

**DESCRIPTION**

Returns a new image of the working size containing a vertical color blend from the start color at the top to the end color at the bottom, in the specified color space.

## WARP(CLIP_IMAGE, WARP_MAP, boolean)

**SYNOPSIS**

```
CLIP_IMAGE WARP(
     CLIP_IMAGE Clip_Image,
     WARP_MAP Warp_Map
     [, boolean Maintain_Aspect ])
```

**PARAMETERS**

| | |
|---|---|
| *Clip_Image* | Source clip image |
| *Warp_Map* | Warp map |
| *Maintain_Aspect* | Optional maintain aspect flag; default value is `false` |

**DESCRIPTION**

Returns a copy of the source clip image warped according to the spatial mapping encoded in the specified warp map (refer to the WARP_MAP data type). The warp map is scaled to fit the source clip image.

The optional *maintain aspect* flag specifies, if `true`, that the aspect ratio of the warp map be maintained. This means that the warp map is cropped to the aspect ratio of the source image. As a result it may contain input coordinates which refer to pixels outside the source image. These coordinates are reflected in the edge of the source image. If the flag is `false`, then the warp map is fitted to the source clip image. If the source clip image and warp map have the same aspect ratio, then the flag has no effect.

## WARP(IMAGE, WARP_MAP, boolean, RECTANGLE_LIST)

**SYNOPSIS**

```
IMAGE WARP(
     IMAGE Image,
     WARP_MAP Warp_Map
     [, boolean Maintain_Aspect ]
     [, RECTANGLE_LIST Regions ])
```

**PARAMETERS**

| | |
|---|---|
| *Image* | Source image |
| *Warp_Map* | Warp map |
| *Maintain_Aspect* | Optional maintain aspect flag; default value is `false` |
| *Regions* | Optional list of target regions |

**DESCRIPTION**

Returns a copy of the source image warped according to the spatial mapping encoded in the specified warp map (refer to the WARP_MAP data type). The warp map is scaled to fit the source image.

The optional *maintain aspect* flag specifies, if `true`, that the aspect ratio of the warp map be maintained. This means that the warp map is cropped to the aspect ratio of the source image. As a result it may contain input coordinates which refer to pixels outside the source image. These coordinates are reflected in the edge of the source image. If the flag is `false`, then the warp map is fitted to the source image. If the source image and warp map have the same aspect ratio, then the flag has no effect.

The optional list of target regions specifies regions in the image to be warped independently using the same warp map. In the absence of a list of target regions, the entire image is warped.

## X_SIZE()

### SYNOPSIS

```
real X_SIZE()
```

### DESCRIPTION

Returns the x component of the working size.

## Y_SIZE()

### SYNOPSIS

```
real Y_SIZE()
```

### DESCRIPTION

Returns the y component of the working size.

# 8 Vark Language Reference

## 8.1 Syntax Notation

The following sections describe Vark syntax using production rules. Syntactic categories are indicated by *italic* type. Literal words and characters are indicated by `constant width bold` type. Alternatives are listed on separate lines, except in a few cases where long lists of alternatives are listed on one line, marked by "one of". *[ A ]* indicates that A is optional. *{ A }* indicates that A occurs zero or more times.

## 8.2 Lexical Structure

There are five kinds of lexical tokens: identifiers, keywords, operators, separators, and constants. White space, consisting of spaces, horizontal tabs, newlines, and comments, is ignored except as it serves to separate symbols. White space is required to separate otherwise adjacent identifiers, keywords and constants.

### 8.2.1 Comments

The characters `/*` start a comment, which terminates with the characters `*/`. These comments do not nest. The characters `//` start a comment, which terminates at the end of the line on which they occur. The comment characters `//`, `/*` and `*/` have no special meaning within a `//` comment. The comment characters `//` and `/*` have no special meaning within a `/*` comment.

### 8.2.2 Identifiers

An identifier consists of an arbitrarily long sequence of letters, digits and underscores. The first character must be a letter. Upper-case and lower-case letters are distinct. All characters are significant.

### 8.2.3 Keywords, Operators, Separators

The following identifiers are reserved for use a keywords, and may not be used otherwise:

```
and             false           or              true
alias           function        procedure       try
array           if              record          type
catch           meta_variable   return          variable
constant        modulo          size            void
do              not             string          while
else            of              then
enumeration     operator        throw
```

The following characters are used as operators or for punctuation:

```
*    (    )    -    +    =    {    }    [    ]
;    '    :    "    <    >    ?    ,    .    /
```

The following character sequences are used as operators:

```
<=   >=   ==   <>
```

### 8.2.4 Constants

Constants specify constant (or literal) values of various types, including numbers, characters and character strings. Constants are described in detail in Section 8.4.3.

## 8.3 PROGRAM STRUCTURE

A Vark program consists of a (possibly empty) set of global declarations, followed by the (possibly empty) body of the main procedure:

*program:*
*global-declarations main-procedure*

*main-procedure:*
*variable-declarations statement-list*

*statement-list:*
*{ statement }*

## 8.4 DECLARATIONS AND CONSTANTS

### 8.4.1 Declarations

A declaration (typically) defines an entity, such as a variable or function, and associates a name (i.e. identifier) with it. The declaration introduces the name into a scope, i.e. a region of the program text where the name is active or visible. A name declared inside a function is in scope from the point at which it is declared to the end of the function (or to the end of the block, if declared within a block). A name declared outside a function is in scope from the point at which it is declared to the end of the program.

#### 8.4.1.1 Global Declarations

In Vark, all types, constants, functions and procedures are declared before the main procedure, and cannot be nested. They therefore have global scope.

*global-declarations:*
*{ global- declaration }*

*global-declaration:*
*type-declaration*
*constant-declaration*
*function-declaration*
*procedure-declaration*

#### 8.4.1.2 Variable Declarations

In Vark, all variables are declared at the start of a function or procedure body. They therefore have function or procedure scope.

*variable-declarations :*
*{* `variable` *type identifier ; }*

A variable name is not allowed to hide a global name.

### 8.4.2 Types

Every constant, variable, operator and function in Vark has a specific type. Vark is strongly typed. Operand types are checked, and there is no implicit type conversion. (There are, however, overloaded operators and functions).

Vark provides a set of primitive types; it provides the means for constructing more complex types via aggregate types; and it provides the means for explicitly naming types.

*type:*
*primitive-type*
*aggregate-type*
*type-identifier*

#### 8.4.2.1 Primitive Types

Vark provides the following primitive types: integer (32-bit signed integer), real (64-bit floating point), boolean (true, false), and character (16-bit unsigned character).

*primitive-type:*
`integer`
`real`
`boolean`
`character`

#### 8.4.2.2 Aggregate Types

Vark provides the means for constructing complex types via the following aggregate aggregate types: array, string and record.

*aggregate-type:*
*array-type*
*string-type*
*record-type*

##### 8.4.2.2.1 Array

An array type defines a sequence of objects of a given type. The size of the sequence is specified and may not vary at run-time. The object type may be primitive or aggregate.

*array-type:*
`array [` *constant-expression* `] of` *type*

##### 8.4.2.2.2 String

A string type defines a sequence of objects of a given type. The size of the sequence is unspecified and may vary at run-time. The object type may be primitive or aggregate.

*string-type:*
`string of` *type*

##### 8.4.2.2.3 Record

A record type defines a sequence of objects of various given types. The size of the sequence is implicitly specified and may not vary at run-time. The object types may be primitive or aggregate.

*record-type:*
`record (` *field-declaration { ; field-declaration }* `)`

*field-declaration:*
*type identifier*

#### 8.4.2.3 Named Types

Separate occurrences of the same type declaration in a program are considered equivalent for the purposes of type checking.

A named type defines a new type which, although structurally equivalent to the type it is based on, is nonetheless considered distinct from the base type for the purposes of type checking.

An aliased type defines a new name for a specified type, but does not define a new type. It simply provides a handy short-hand for the base type.

An enumerated type defines a named type based on `integer`, and a set of named constants of that type with integer values starting at zero and increasing by one.

*type-declaration:*
`type` *type identifier ;*

**alias** *type identifier* ;
**enumeration** ( *identifier* { , *identifier* } ) *identifier* ;

### 8.4.3 Constants

There is a constant format corresponding to each primitive type.

*constant:*
*integer-constant*
*real-constant*
*boolean-constant*
*character-constant*
*character-string-constant*

#### 8.4.3.1 Integer Constant

A decimal integer constant consists of a sequence of the digits decimal digits (`0` through `9`). An octal integer constant consists of the prefix characters `0o` or `0O` followed by a sequence of octal digits (`0` through `7`). A hexadecimal constant consists of the prefix `0x` or `0X` followed by a sequence of hexadecimal digits (`0` through `9`, and `A` through `F`, or `a` through `f`, representing decimal 10 through 15). For example, decimal twelve can be written `12`, `0o14`, or `0xC`.

#### 8.4.3.2 Real Constant

A real constant consists of an integer part, a decimal point, a fraction part, and an exponent. The integer and fraction parts both consist of a sequence of decimal digits. Either the integer part or the fraction part may be missing, but not both. The exponent consists of `e` or `E`, an optional sign, and an integer exponent value. Either the decimal point or the exponent may be missing, but not both. For example, twelve can be written `12.`, `1.2e1`, `.12e2`, `120e-1`.

#### 8.4.3.3 Boolean Constant

There are two predefined boolean constants `true` and `false`.

#### 8.4.3.4 Character Constant

A character constant consists of a character enclosed in single quotes. Certain non-graphic characters may be represented according to the following table of escape sequences:

| | |
|---|---|
| single quote | `\'` |
| double quote | `\"` |
| backslash | `\\` |
| newline | `\n` |
| horizontal tab | `\t` |
| octal number | `\o`*nnnnnn* or `\O`*nnnnnn* |
| decimal number | `\d`*nnnnn* or `\D`*nnnnn* |
| hexadecimal number | `\x`*nnnn* or `\X`*nnnn* |

The number escape sequences contain at least one digit. The octal number sequence contains at most six digits, the decimal sequence at most five digits, the hexadecimal sequence at most four digits. The largest allowed character value is $2^{16} - 1$.

#### 8.4.3.5 Character String Constant

A character string constant consists of sequence of characters enclosed in double quotes. Character escape sequences may be embedded in the sequence, specified in the same way as for character constants.

### 8.4.4 Named Constants and Meta-Variables

A named constant associates a name with a constant value at parse-time.

*constant-declaration:*
`constant` *type identifier* = *constant-expression* ;

A meta-variable also associates a name with a constant value at parse-time, but the interpreter is free to alter the value, in some environment-specific way, before executing the program. Since the value associated with a meta-variable is constant with respect to the program, it behaves just like a named constant in the program.

*meta-variable-declaration:*
`meta_variable` *type identifier* = *constant-expression* ;

A constant expression is any expression which can be evaluated at parse-time.

## 8.5 Expressions and Statements

A function (or procedure) body consists of a sequence of statements which are executed in order. There are two kinds of statements: those that modify the function's variables, and those that control the flow of execution.

Statements evaluate expressions – to determine the value to assign to a variable, or to determine the value of a condition which controls the flow of execution.

An expression specifies a value directly (as a constant or variable reference) or as a computation in the form of an operator (or function) and a set of operands (or parameters). An expression evaluates to a single value. An expression does not have any side effects.

### 8.5.1 Variable Reference

A variable reference may appear as an expression, where the value of the variable is read; and on the left-hand-side of an assignment statement, where the value of the variable is written.

There are three kinds of variable references:

*variable:*
*simple-variable*
*indexed-variable*
*field-variable*

A simple variable reference refers to an entire named variable.

An indexed variable reference refers to an element of an array variable or string variable, indexed by the value of the index expression (which must be of type `integer`). The index is zero-based. It must be in the range of the size of the array or string.

*indexed-variable:*
*array-or-string-variable* [ *expression* ]

A field variable reference refers to a named field within a record variable:

*field-variable:*
*record-variable* . *field-identifier*

Note that the indexing ([]) and field reference (.) operators only operate on variables, not on general expressions.

### 8.5.2 Expressions

Every expression has a type. When the expression contains an operator or function call, the type of the expression is determined by the operator or function name and the types of the operands. Operators and functions may be overloaded, but unless an operator or

function of the specified name is defined for the specified operand types, the expression is invalid. There is no implicit type conversion of operands.

The operand expressions are evaluated before the operator or function is executed.

#### 8.5.2.1 Operators

Operators fall into the following five categories: relational, arithmetic, string, logical, and type.

##### 8.5.2.1.1 Relational Operators:

| | |
|---|---|
| equal | *expression* == *expression* |
| not equal | *expression* <> *expression* |
| less than | *expression* < *expression* |
| less than or equal | *expression* <= *expression* |
| greater than | *expression* > *expression* |
| greater than or equal | *expression* >= *expression* |

Built-in relational operators are defined for types `integer` and `real`. They return a value of type `boolean`.

Built-in equality and inequality operators are defined for all types. For aggregate types, they are defined recursively.

##### 8.5.2.1.2 Arithmetic Operators

| | |
|---|---|
| add | *expression* + *expression* |
| subtract | *expression* - *expression* |
| multiply | *expression* * *expression* |
| divide | *expression* / *expression* |
| remainder | *expression* `modulo` *expression* |
| negate | - *expression* |

Built-in arithmetic operators are defined for types `integer` and `real`. They return a value of the same type as their operands.

Unary plus (+) is defined, but has no effect.

##### 8.5.2.1.3 String Operators

| | |
|---|---|
| string size | `size` *expression* |
| concatenate strings | *expression* + *expression* |
| substring | *variable* [ *expression* , *expression* ] |
| construct string | [ [ *expression* { , *expression* } ] ] |

The built-in `size` operator returns the size of the array or string, as an `integer`.

The built-in concatenation operator returns a string of the same type as its string operands, containing the concatenation of its string operands. The two string operands must be of the same type.

The built-in substring operator returns a string of the same type as its string operand, containing the substring of the string operand identified by the two index expressions. The index expressions must be of type `integer`. The indices must be in the range of the string size, and equal or ascending.

The built-in string constructor operator returns a string whose object type is the type of its operands, containing the values of its operands. Every operand must have the same type. The empty string ([]) has type `string of void`, and must be cast before it can be used (see below).

#### 8.5.2.1.4 Logical Operators

| | |
|---|---|
| logical and | *expression* `and` *expression* |
| logical or | *expression* `or` *expression* |
| logical not | `not` *expression* |

The built-in logical operators are defined for type `boolean`. They return a value of type `boolean`. The `and` and `or` operators always evaluate both their arguments.

The `and` operator returns the logical and of its operands. The `or` operator returns the logical or of its operands. The `not` operator returns the logical not (or inverse) of its operand.

#### 8.5.2.1.5 Type Operators

| | |
|---|---|
| cast | *type* : *expression* |

The built-in cast operators are defined for converting between `integer` and `real`, between a named type and its base type, and from the type of the empty string (`string of void`) to any string type. They return the value of the expression, converted to the specified type.

### 8.5.2.2 Function Calls

A function call expression consists of a function identifier followed by a parenthesised list of parameter bindings. The type of the expression is determined by the function name and the types of the parameter expressions, as described earlier.

Each parameter binding optionally names the corresponding formal parameter. If the formal parameter is named, then the name is verified.

The function call binds the evaluated parameter expressions to the function's formal parameters and executes the body of the function. The value of the function is the value of the expression which the function returns via a `return` statement. All function parameters are passed by value, so a function cannot cause side effects by modifying its parameters.

*function-call-expression:*
*function-identifier* ( *parameter-bindings* )

*parameter-bindings:*
*[ parameter-binding { , parameter-binding } ]*

*parameter-binding:*
*[ formal-parameter-identifier = ] expression*

Functions are discussed in greater detail in a later section.

### 8.5.2.3 Constants and Variables

A constant forms an expression whose type and value are the type and value of the constant.

A variable reference forms an expression whose type and value are the type and value of the variable.

### 8.5.2.4 Evaluation Order

Operators have a defined precedence and associativity. When more than one operator appears in an expression, the operators are evaluated according to their precedence and associativity. Parentheses may be used to specify a different order of evaluation, or simply to make the default order obvious.

Operators have the following precedence (from high precedence to low):

| | |
|---|---|
| 1 | `[] . ()` |
| 2 | + (unary), - (unary), `not`, `size` |
| 3 | `:` |
| 4 | `* / modulo and` |
| 5 | `+ - or` |
| 6 | `== <> < <= > >=` |

Unary operators associate right-to-left. All other operators associate left-to-right.

For example, `2 * 3 + 4 * 5` is equivalent to `(2 * 3) + (4 * 5)`, because * has higher precedence than +. The addition can be given precedence by using parentheses: `2 * (3 + 4) * 5`. And `2 * 3 / 4` is equivalent to `(2 * 3) / 4` because * and / associate left-to-right.

### 8.5.3 Statements

As described earlier, there are two kinds of statements: those that modify the function's variables, and those that control the flow of execution.

The assignment statement and procedure call statement fall into the former category, and the `if`, `while`, `return` and `throw` statements fall into the latter.

*statement:*
*assignment-statement*
*if-statement*
*while-statement*
*procedure-call-statement*
*return-statement*
*throw-statement*
*compound-statement*

A compound statement consists of a sequence of statements grouped so as to act as a single statement:

*compound-statement :*
{ *statement-list* }

#### 8.5.3.1 Assignment Statement

An assignment statement assigns the value of the specified expression to the specified variable. The type of the expression must match the type of the variable.

*assignment-statement:*
*variable* = *expression* ;

#### 8.5.3.2 If Statement

An `if` statement evaluates its condition expression, and if it evaluates `true` executes the statement following the `then`. Otherwise it executes, if present, the statement following the `else`. The condition expression must of type `boolean`.

*if-statement:*
`if` *expression* `then` *statement* [ `else` *statement* ]

#### 8.5.3.3 While Statement

A `while` statement repeatedly evaluates its condition expression, and every time it evaluates `true` executes the statement following the `do`. It continues in this fashion until the condition expression evaluates `false`. The condition expression must of type `boolean`.

*while-statement:*
`while` *expression* `do` *statement*

#### 8.5.3.4 Procedure Call Statement

A procedure call statement consists of a procedure identifier followed by a parenthesised list of parameter bindings.

Each parameter binding optionally names the corresponding formal parameter. If the formal parameter is named, then the name is verified.

The procedure call binds the evaluated parameter expressions to the procedure's formal parameters and executes the body of the procedure. Unlike a function call, a procedure call does not return a value. Also, unlike a function call, a procedure call may have side effects: a procedure parameter may be declared as `variable`, in which case the actual parameter, which must be a variable reference, is passed by reference rather than value.

*procedure-call-statement:*
*procedure-identifier* ( *parameter-bindings* ) ;

*parameter-bindings:*
[ *parameter-binding* { , *parameter-binding* } ]

*parameter-binding:*
[ *formal-parameter-identifier* = ] *expression*

Procedures are discussed in greater detail in a later section.

#### 8.5.3.5 Return Statement

A `return` statement returns control from the current function or procedure to the calling function or procedure. A `return` statement in a function also specifies an expression whose value becomes the value of the function. The type of the expression must match the type of the function.

*return-statement:*
**`return`** [ *expression* ] ;

#### 8.5.3.6 Throw Statement

A `throw` statement generates an exception of the type and value of the expression. A `throw` statement passes control to the latest active exception handler for that type, which may reside in some calling function or procedure.

*throw-statement:*
**`throw`** *expression* ;

Exception handling is discussed in greater detail in a later section.

## 8.6 Functions and Procedures

### 8.6.1 Functions

A function consists of a function heading, which defines the function's interface, and a function body which defines the function's implementation.

*function-declaration:*
*function-heading function-body*

The function heading specifies the function's return type and name, and the types and names of its formal parameters.

*function-heading:*
**`function`** *type identifier* ( *func-parameter-declarations* )

*func-parameter-declarations:*
*[ func-parameter-declaration { , func-parameter-declaration } ]*

*func-parameter-declaration:*
*type identifier*

The function body defines the variables and statements which implement the function's algorithm.

*function-body:*
{ *variable-declarations statement-list* }

### 8.6.2 Procedures

A procedure consists of a procedure heading, which defines the procedure's interface, and a procedure body which defines the procedure's implementation.

*procedure-declaration:*
*procedure-heading procedure-body*

The procedure heading specifies the procedure's name, and the types and names of its formal parameters. Note that, unlike functions, procedures are allowed `variable` parameters which are passed by reference rather than by value.

*procedure-heading:*
**procedure** *identifier* ( *proc-parameter-declarations* )

*proc-parameter-declarations:*
*[ proc-parameter-declaration { , proc-parameter-declaration } ]*

*proc-parameter-declaration:*
*[* **variable** *] type identifier*

The procedure body defines the variables and statements which implement the procedure's algorithm.

*procedure-body:*
{ *variable-declarations statement-list* }

### 8.6.3 Overloading

Both functions and procedures may be overloaded: i.e. multiple functions and procedures may share the same name so long as their parameters differ in number and/or type.

Overloaded functions and procedures support an object-oriented perspective – a function (or operator, see below) which has the same meaning for different types can have the same name when implemented for those different types.

Overloaded functions and procedures also indirectly support default arguments – just supply a different function or procedure for each desired subset of the parameter list.

Each function and procedure has a unique signature made up of its name and the types of its parameters, in the order they appear in the formal parameter list. Since Vark is a strongly typed language and performs no implicit type conversion, it is easy to identify, from the name and actual parameter expression types appearing in the function or procedure call, the correct version of a function or procedure to execute.

Note that the return type of a function is not part of the function's signature, and the `variable` declaration of a procedure parameter is not part of the parameter's type, and so is not part of the procedure's signature either. However, since a function call expression is distinct from a procedure call statement, function signatures are considered distinct from procedure signatures. Thus a procedure may overload a function which has the same name *and* parameter list.

A procedure is typically used to overload a function which returns a modified version of one of its arguments. The procedure typically modifies the corresponding argument directly. In the absence of such a procedure, Vark constructs one. For a function of the form:

```
function type name ( ..., type parameter, ... )
```

Vark constructs a procedure of the form:

```
procedure name ( ..., variable type parameter, ... )
{
    parameter = name ( ..., parameter, ... )
}
```

The variable parameter is taken to be the first parameter with the same type as the function's return type.

### 8.6.4 Operators

Most operators may also be overloaded.

*function-heading:*
`operator` *type overloadable-operator* `(` *func-parameter-declarations* `)`

*overloadable-operator:* one of
`== <> < <= > >= + - * / modulo and or not :`

The precedence and associativity of an operator cannot be changed, nor can the number of operands it expects. For most operators there is a natural relationship between their operand types and return types. These relationships need not be maintained in overloaded versions of operators, but in most cases probably should be.

## 8.7 EXCEPTION HANDLING

With the exception of the `throw` statement, exception handling is not yet supported in Vark. Exceptions thrown within a Vark program currently propagate to a handler in the interpreter.

# 9 Vark Syntax Summary

*program:*
*global-declarations main-procedure*

*global-declarations:*
*{ global-declaration }*

*global-declaration:*
*type-declaration*
*constant-declaration*
*meta-variable-declaration*
*function-declaration*
*procedure-declaration*

*type-declaration:*
`type` *type identifier ;*
`alias` *type identifier ;*
`enumeration` `(` *identifier { , identifier }* `)` *identifier ;*

*constant-declaration:*
`constant` *type identifier* `=` *constant-expression ;*

*meta-variable-declaration:*
`meta_variable` *type identifier* `=` *constant-expression ;*

*function-declaration:*
*function-heading function-body*

*function-heading:*
`function` *type identifier* `(` *func-parameter-declarations* `)`
`operator` *type overloadable-operator* `(` *func-parameter-declarations* `)`

*overloadable-operator:* one of
`==` `<>` `<` `<=` `>` `>=` `+` `-` `*` `/` `modulo` `and` `or` `not` `:`

*func-parameter-declarations:*
*[ func-parameter-declaration { , func-parameter-declaration } ]*

*func-parameter-declaration:*
*type identifier*

*function-body:*
`{` *variable-declarations statement-list* `}`

*procedure-declaration:*
*procedure-heading procedure-body*

*procedure-heading:*
`procedure` *identifier* `(` *proc-parameter-declarations* `)`

*proc-parameter-declarations:*
*[ proc-parameter-declaration { , proc-parameter-declaration } ]*

*proc-parameter-declaration:*
*[* `variable` *] type identifier*

*procedure-body:*
`{` *variable-declarations statement-list* `}`

*type:*
*primitive-type*
*aggregate-type*
*type-identifier*

*primitive-type:*
**integer**
**real**
**boolean**
**character**

*aggregate-type:*
*array-type*
*string-type*
*record-type*

*array-type:*
**array** [ *constant-expression* ] **of** *type*

*string-type:*
**string of** *type*

*record-type:*
**record** ( *field-declaration* { ; *field-declaration* } )

*field-declaration:*
*type identifier*

*type-identifier:*
*identifier*

*variable-declarations :*
{ **variable** *type identifier* ; }

*main-procedure:*
*variable-declarations statement-list*

*statement-list:*
{ *statement* }

*statement:*
*assignment-statement*
*if-statement*
*while-statement*
*procedure-call-statement*
*return-statement*
*throw-statement*
*compound-statement*

*assignment-statement:*
*variable* = *expression* ;

*variable:*
*simple-variable*
*indexed-variable*
*field-variable*

*simple-variable:*
*variable-identifier*

*variable-identifier:*
*identifier*

*indexed-variable:*

*array-or-string-variable* [ *expression* ]

*array-or-string-variable:*
*variable*

*field-variable:*
*record-variable* . *field-identifier*

*record-variable:*
*variable*

*field-identifier:*
*identifier*

*if-statement:*
`if` *expression* `then` *statement* [ `else` *statement* ]

*while-statement:*
`while` *expression* `do` *statement*

*procedure-call-statement:*
*procedure-identifier* ( *parameter-bindings* ) ;

*procedure-identifier:*
*identifier*

*parameter-bindings:*
[ *parameter-binding* { , *parameter-binding* } ]

*parameter-binding:*
[ *formal-parameter-identifier* = ] *expression*

*formal-parameter-identifier:*
*identifier*

*return-statement:*
`return` [ *expression* ] ;

*throw-statement:*
`throw` *expression* ;

*compound-statement :*
{ *statement-list* }

*constant-expression:*
*expression*

*expression:*
*rel-expression*

*rel-expression:*
*add-expression* [ *rel-operator* *add-expression* ]

*rel-operator:* one of
`==` `<>` `<` `<=` `>` `>=`

*add-expression:*
[ *add-expression* *add-operator* ] *mul-expression*

*add-operator:* one of
`+` `-` `or`

*mul-expression:*
[ *mul-expression* *mul-operator* ] *cast-expression*

*mul-operator:* one of
`*` `/` `modulo` `and`

*cast-expression:*
*unary-expression*
*type* `:` *cast-expression*

*unary-expression:*
*factor*
*unary-operator unary-expression*

*unary-operator:* one of
`+` `-` `not`

*factor:*
*constant*
*variable*
*substring-expression*
*function-call-expression*
*string-constructor-expression*
`(` *expression* `)`

*constant:*
*integer-constant*
*real-constant*
*boolean-constant*
*character-constant*
*character-string-constant*

*substring-expression:*
*variable* `[` *expression* `,` *expression* `]`

*function-call-expression:*
*function-identifier* `(` *parameter-bindings* `)`

*function-identifier:*
*identifier*

*string-constructor-expression:*
`[` [ *expression* { `,` *expression* } ] `]`

# 10 File Formats

The following file formats support Vark's image processing data types *in the demonstration system.*

All TIFF files have the following common characteristics:

- 8 bits per channel
- no compression
- planar configuration
- `.tif` file suffix

The image size (in pixels) and the image print resolution (in pixels per inch) together define the size of the image (in inches) when printed. The print resolution is proportional to the working resolution (pixels per model unit).

## 10.1 Brush Stroke Segment File

A three-channel RGB TIFF.

The red channel contains the segment matte. The green channel contains the segment bump map. The blue channel contains the segment footprint.

## 10.2 Bump Map File

A one-channel grayscale (min-is-black) TIFF.

## 10.3 Clip Image File

A four-channel RGBA TIFF.

The color channels are not pre-multiplied by the matte.

The axis point cannot be specified, and so defaults to the center of the image.

## 10.4 Color Lookup Table (CLUT) File

An Adobe Photoshop arbitrary map settings file (`.amp` file suffix).

Photoshop creation sequence:

*Image → Adjust → Curves → Pencil (Arbitrary Map) → Save*

## 10.5 3D Color Lookup Table (3D CLUT) File

A three-channel RGB TIFF.

The image represents a discrete sampling of an arbitrary mapping function between points in the RGB color space. The image is organised into square tiles of size $2^n + 1$, with $2^n + 1$ tiles arranged vertically. Each tile represents a green-blue slice through the transformed input color space for a particular red value. I.e. the output RGB value for a particular input RGB value is stored in the tile indexed vertically by input red, at a position within the tile indexed vertically by input green and horizontally by input blue. The $\log_2$ of the sampling rate (i.e. n) must be in the range 1 to 8. It is typically 4 or 5.

A gamut compression 3D CLUT can be generated using the following Vark function:

```
CLUT_3D GAMUT_COMPRESSION_CLUT_3D(
    PALETTE Source_Gamut,
    PALETTE Target_Gamut,
    COLOR Luma_Axis_Start,
    COLOR Luma_Axis_End,
    integer Sample_Count)
```

A color morph 3D CLUT can be generated using the following Vark function:

```
CLUT_3D COLOR_MORPH_CLUT_3D(
    PALETTE Source_Palette,
    PALETTE Target_Palette,
    real Scale,
    integer Sample_Count)
```

## 10.6 Displacement Map File

A three-channel RGB TIFF.

The red channel contains y displacement factors. The green channel contains x displacement factors. The blue channel is unused. The color component range 0 to 255 maps to the displacement factor range –1.0 to 1.0.

## 10.7 Dither Matrix File

A one-channel grayscale (min-is-black) TIFF.

## 10.8 Font

A TrueType outline font known to the operating system.

## 10.9 Image File

A three-channel or four-channel RGB TIFF.

The fourth channel, if present, is taken to be a bump map channel.

## 10.10 Image Font File

A 2D image font file (`.2df` file suffix).

Refer to the `Make2DFont` utility user guide [TBA].

## 10.11 Image Font 3D File

A 3D image font file (`.3df` file suffix).

Refer to the `Make3DFont` utility user guide [TBA].

## 10.12 Kernel File

An Adobe Photoshop custom filter file (`.acf` file suffix).

Photoshop cration sequence:

*Filter → Custom → Save*

## 10.13 Matte File

One-channel grayscale (min-is-black) TIFF.

## 10.14 Palette File

An Adobe Photoshop color table file (`.act` file suffix).

Photoshop creation sequence:

*Image → Mode → Indexed Color*
*Image → Mode → Color Table → Save*

## 10.15 Path File

A text file containing path construction commands of the following form:

*Close Path*:
```
0
```
*Move To*:
```
1 x y
```
*Line To*:
```
2 x y
```
*Curve To*:
```
3 x1 y1 x2 y2 x3 y3
```

## 10.16 Tile File

A three-channel RGB TIFF.

The red channel contains the tile matte. The green channel contains the tile bump map. The blue channel contains the tile footprint (which is currently unused).

## 10.17 Vector Map File

A three-channel RGB TIFF.

The red channel contains the direction angle. The green channel contains the magnitude. The blue channel is unused. The red color component range 0 to 255 maps to the angle range 0.0 to 360.0 degrees. The green color component range 0 to 255 maps to the magnitude range 0.0 to 1.0.

## 10.18 Warp Map File

A three-channel RGB TIFF.

The red channel contains x address reverse mapping. The green channel contains y address reverse mapping. The blue channel is unused.

Map dimension should be a power of two, less than or equal to $2^8$ (256).

Addresses in the map should range from 0 to $2^n - 1$, where $2^n$ is the map dimension.

# 11 VarkShop Reference

VarkShop is a Windows 95/NT application used for editing and executing Vark programs. VarkShop contains a fully functional Vark interpreter which conforms to the Vark language specification in this document.

VarkShop installs itself as a default editor for Vark files. A Vark file is a text file which contains a Vark program, and is identified by having a `.vark` file suffix.

# 12 VarkShow Reference

VarkShow is a Windows 95/NT application used for executing slideshows of multiple Vark programs on a particular photo. Varkshow includes the capability to capture a photo from an attached Kodak DC120 camera. VarkShow contains a fully functional Vark interpreter which conforms to the Vark language specification in this document.

VarkShow installs itself as a default editor for Vark slideshow files. A Vark slideshow file is a text file which contains a list of pathnames of Vark programs, and is identified by having a `.varkshow` file suffix.

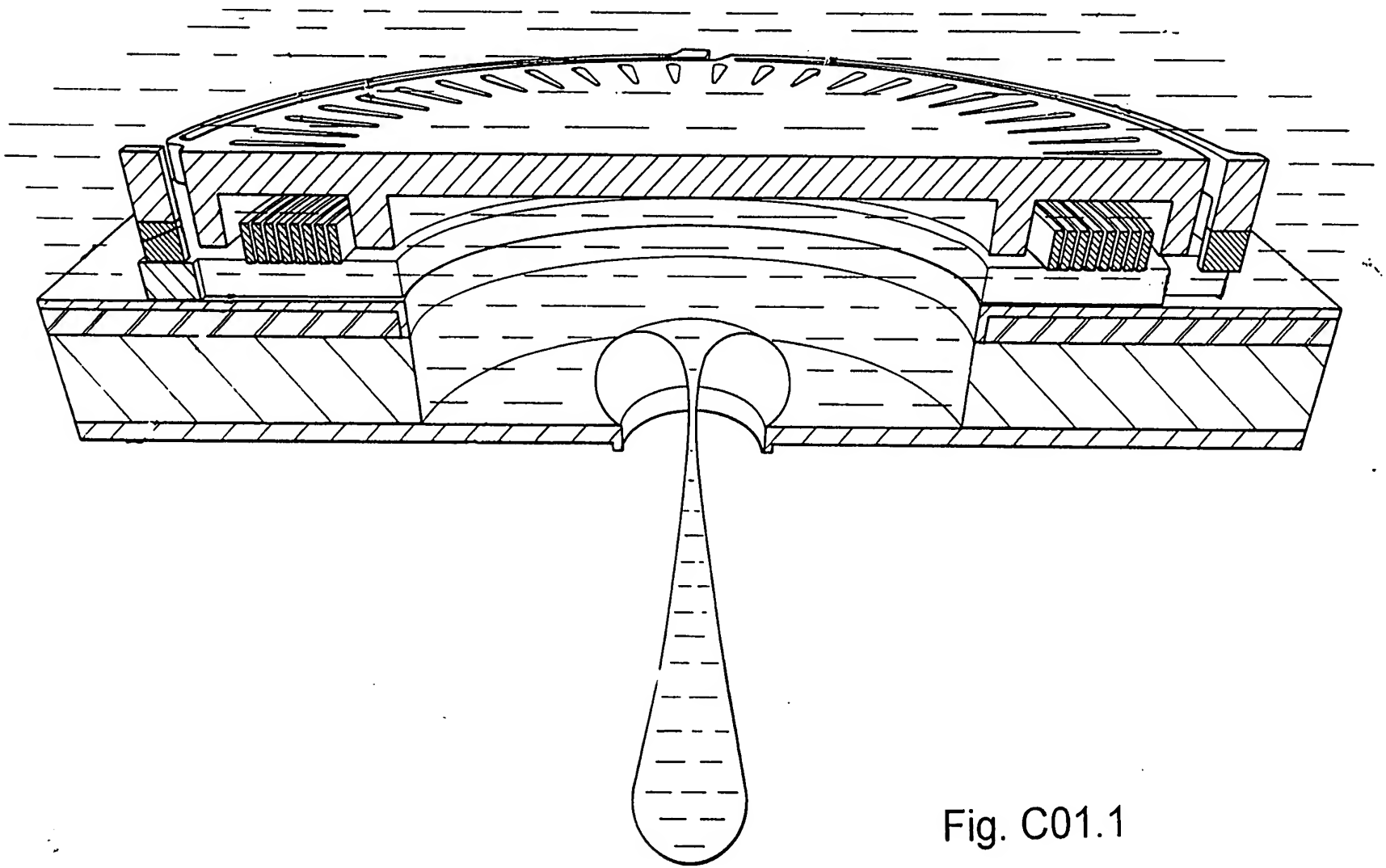


Fig. C01.1

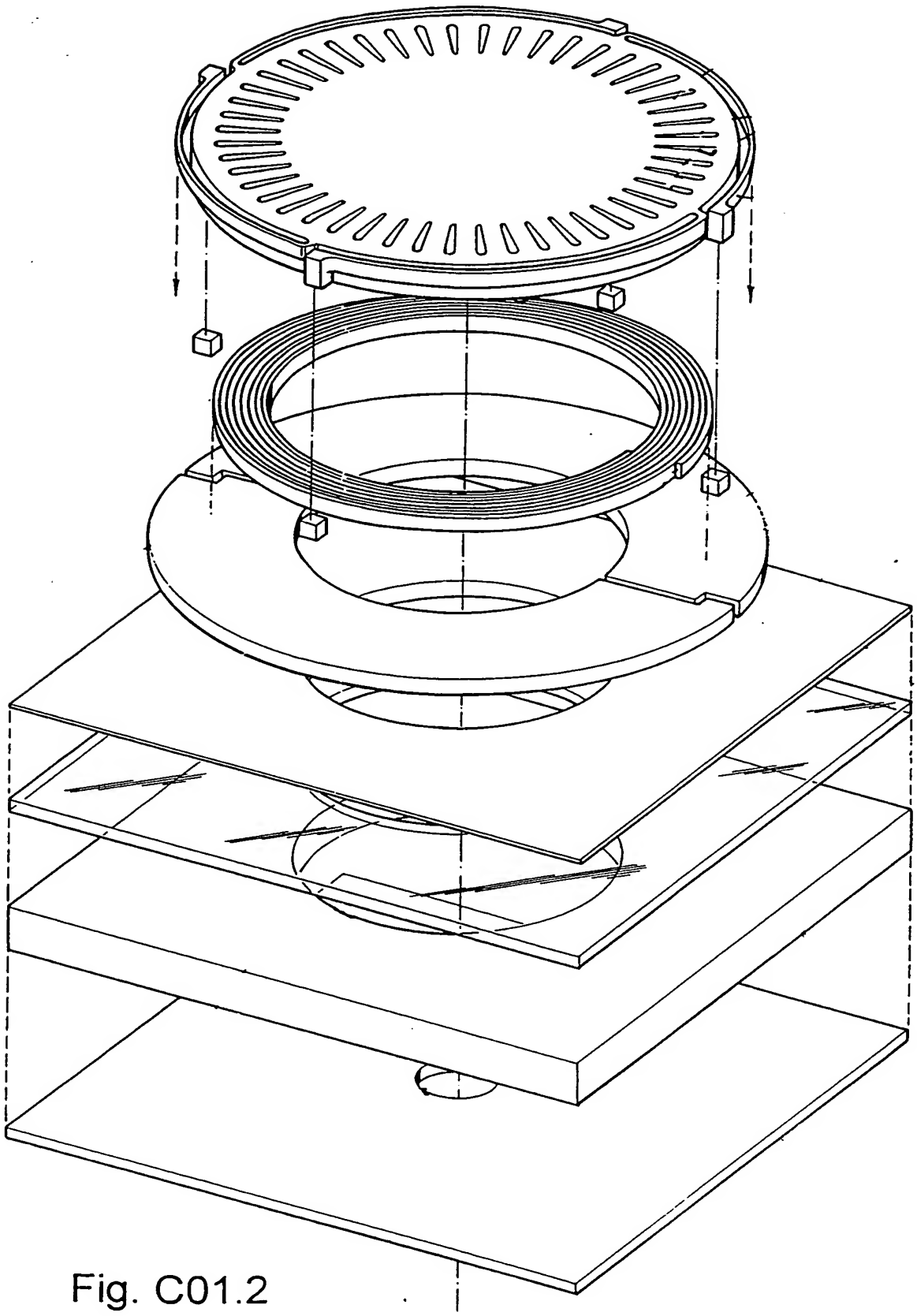


Fig. C01.2

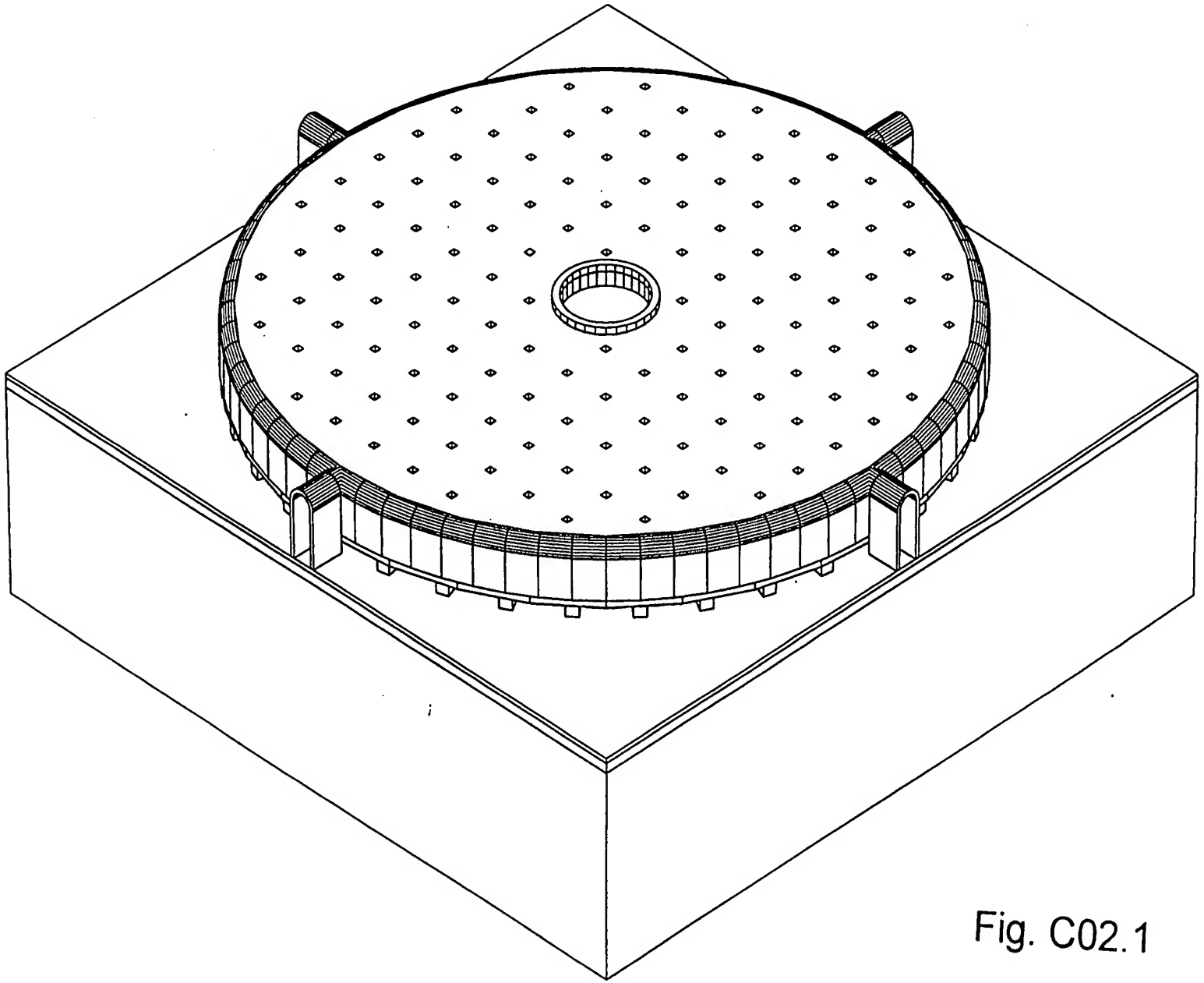


Fig. C02.1

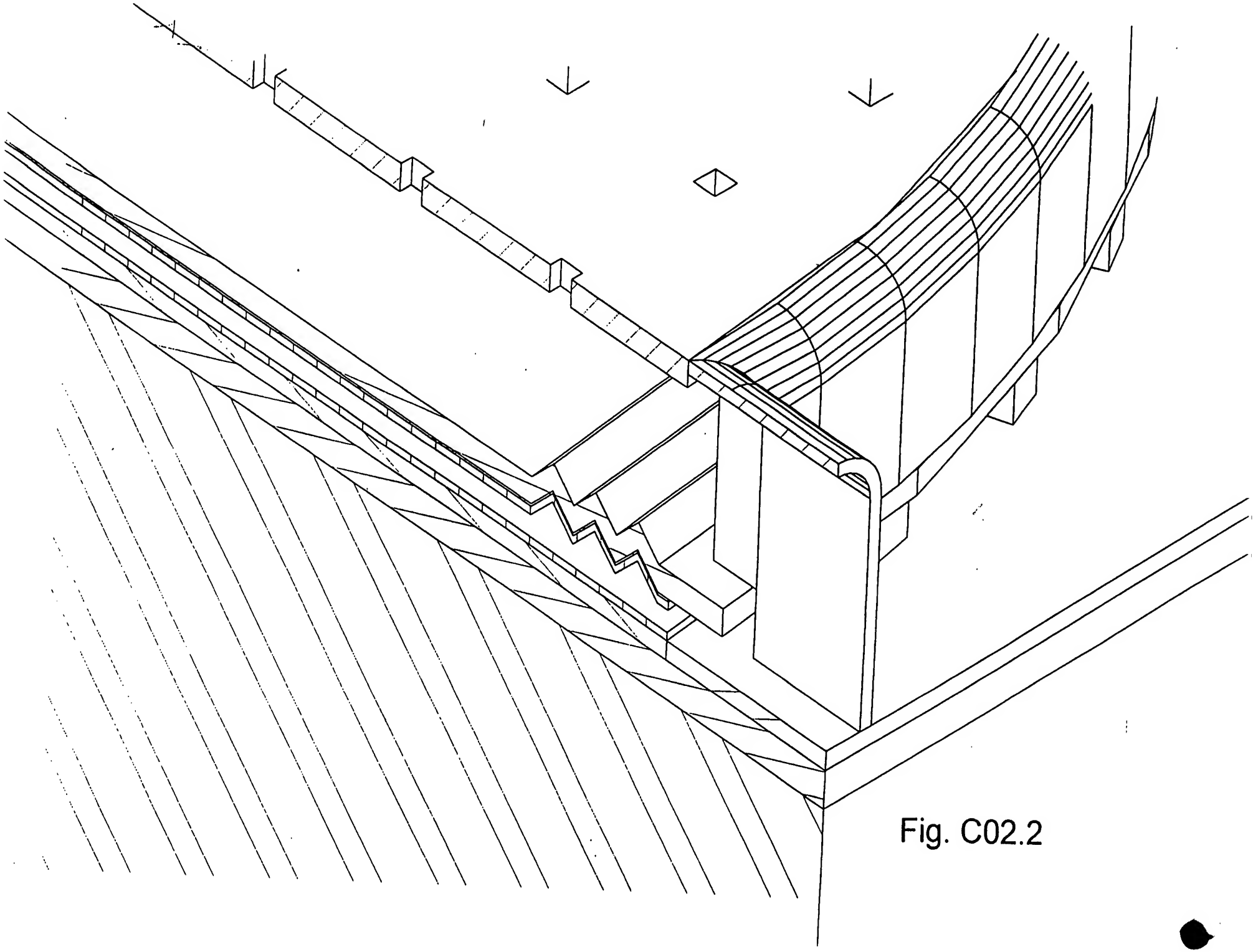


Fig. C02.2

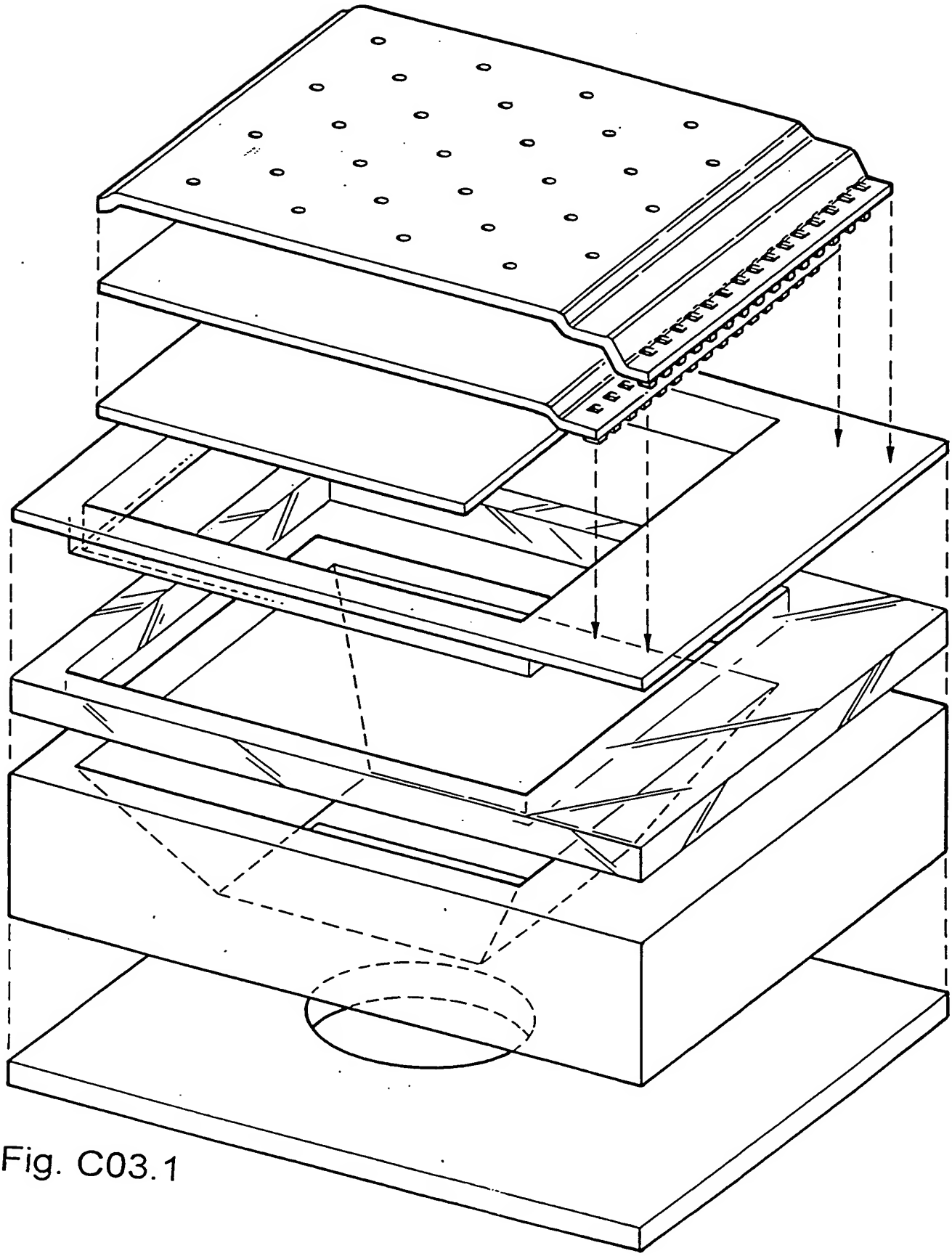


Fig. C03.1

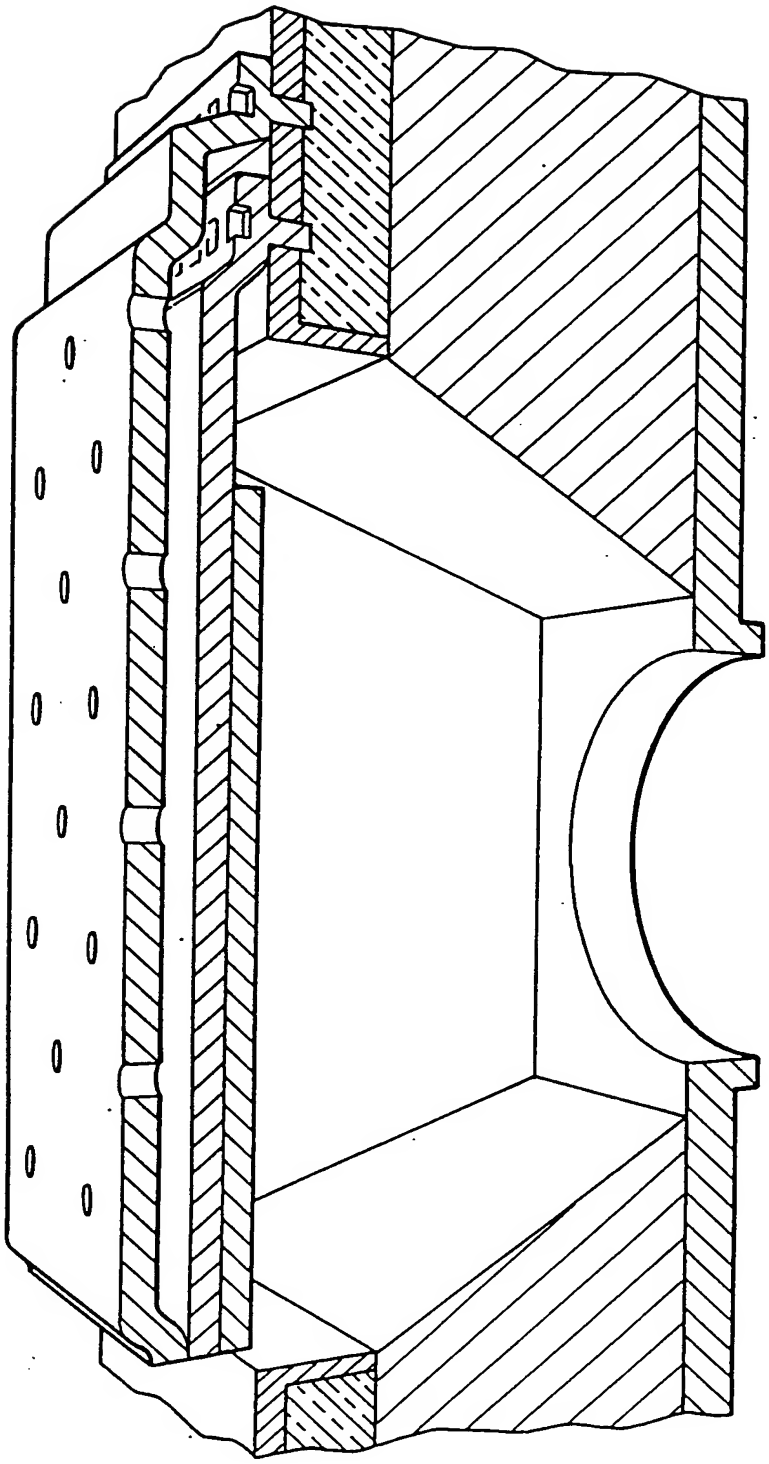


Fig. C03.2

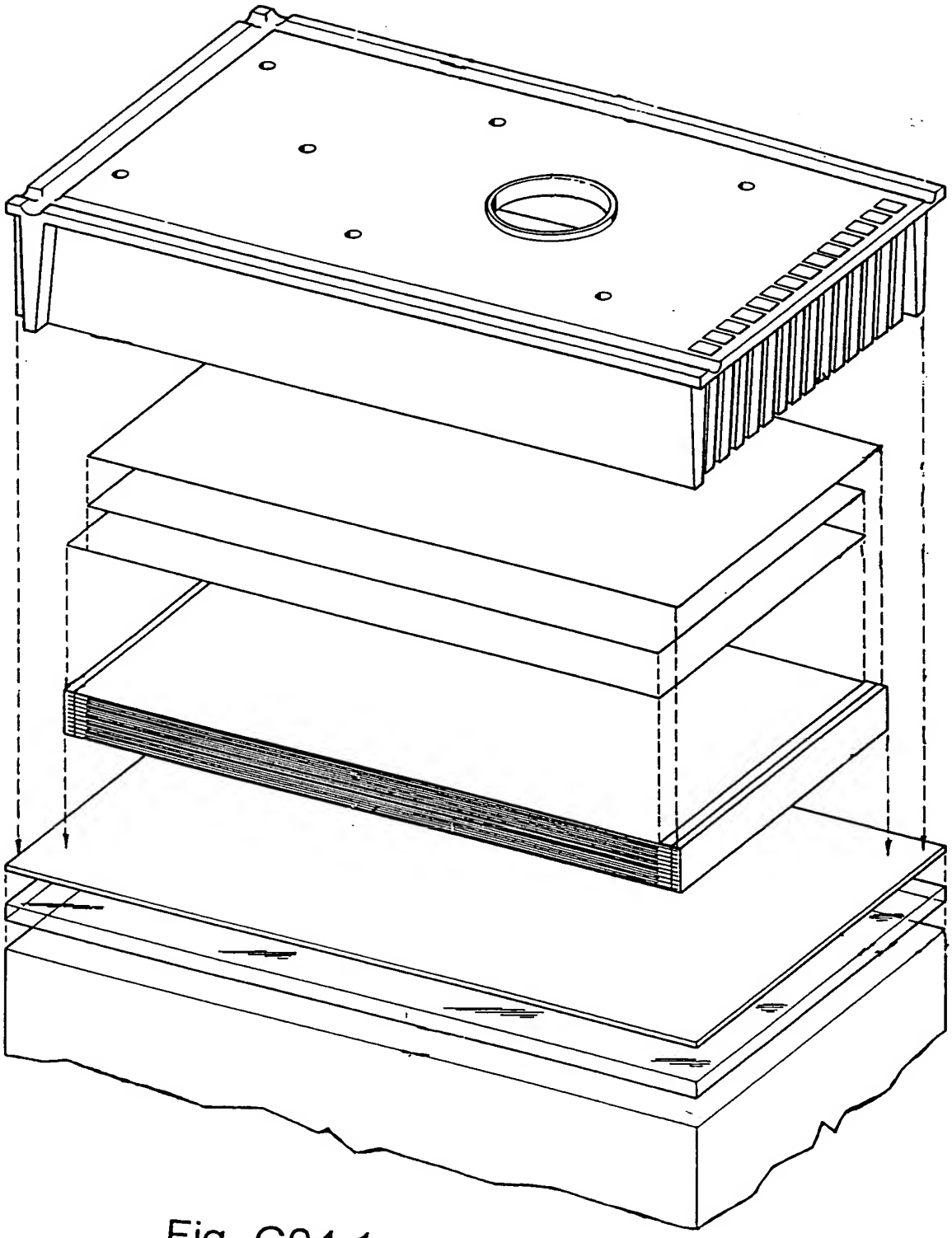


Fig. C04.1

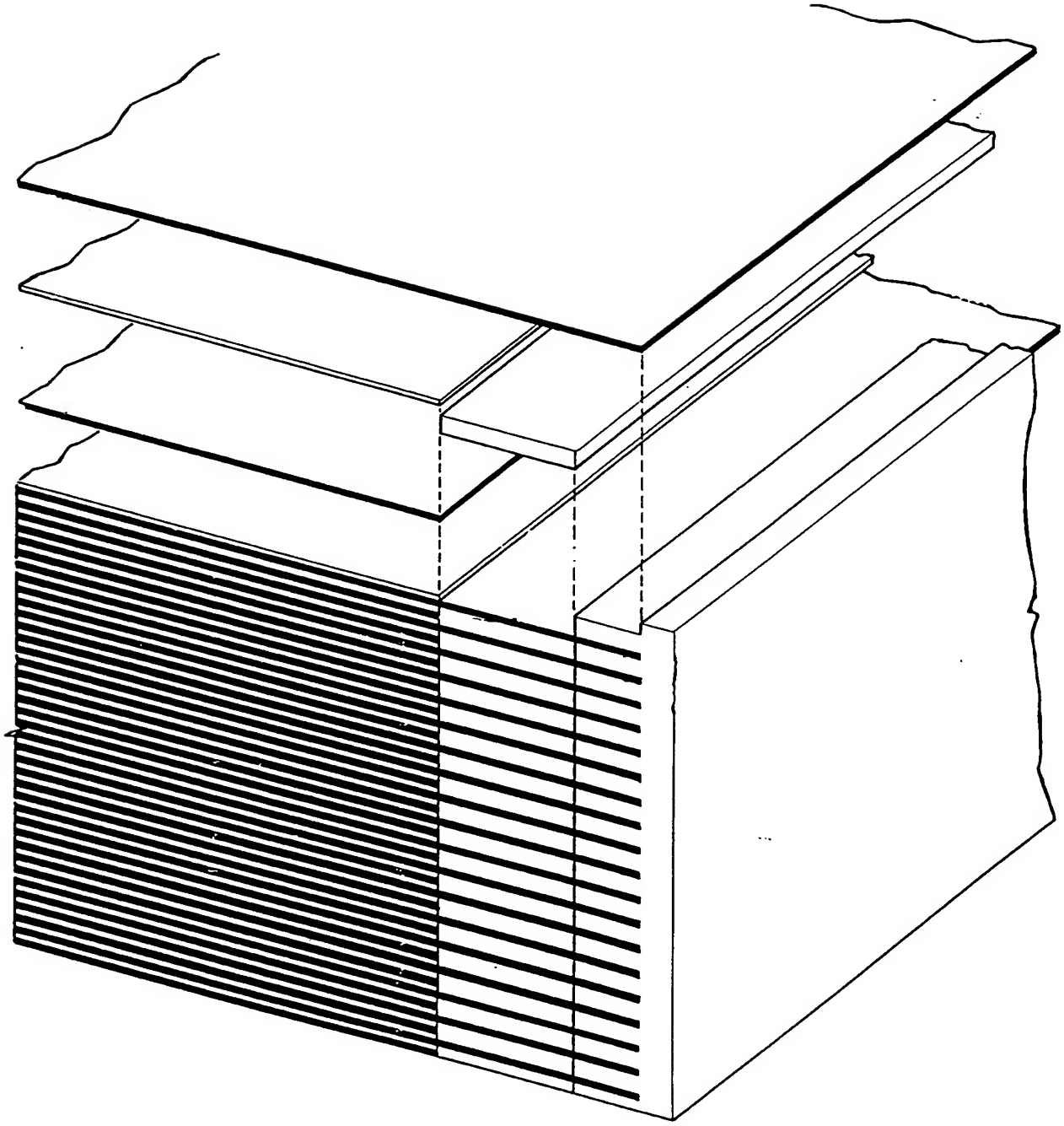


Fig. C04.2

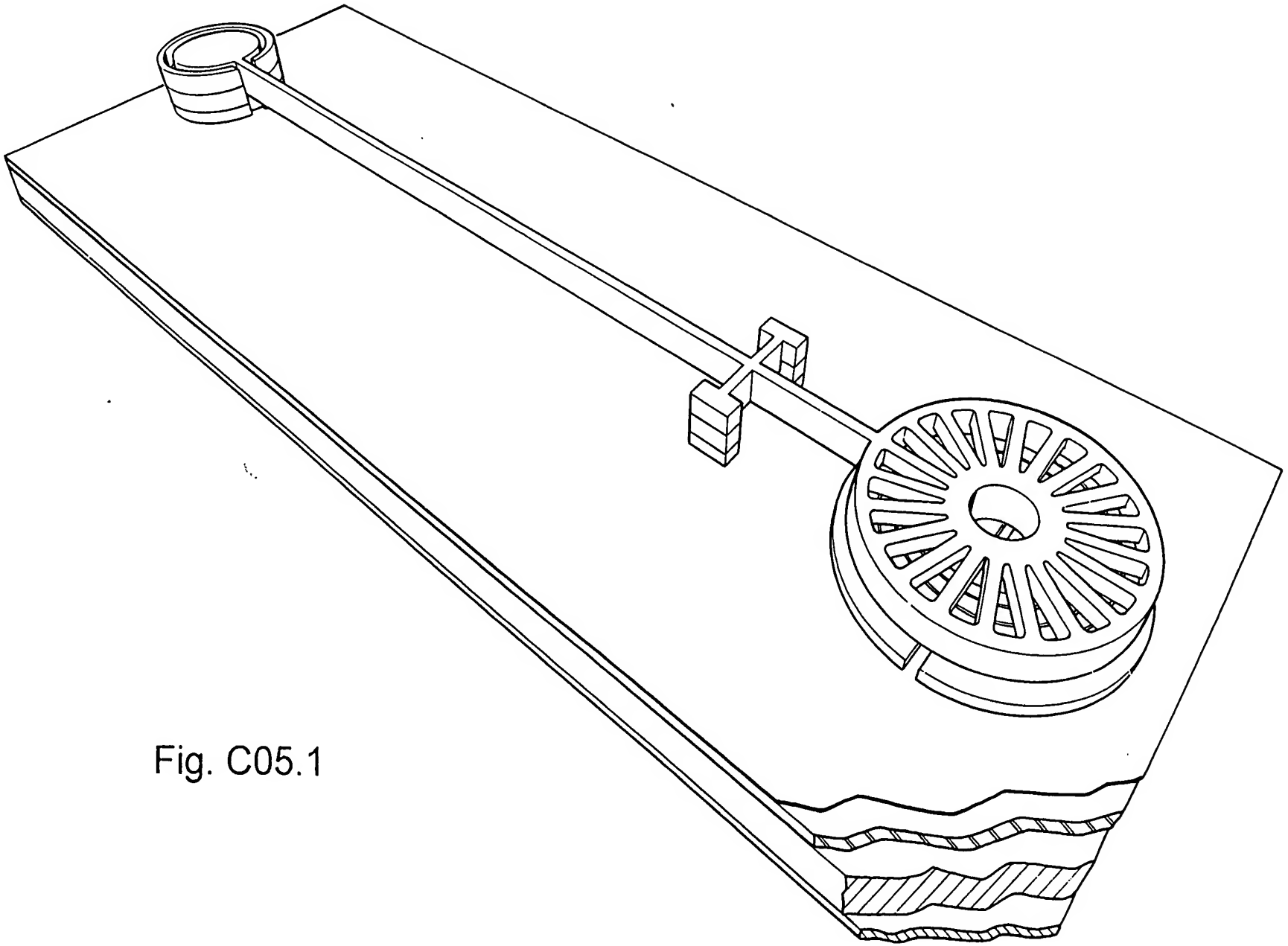


Fig. C05.1

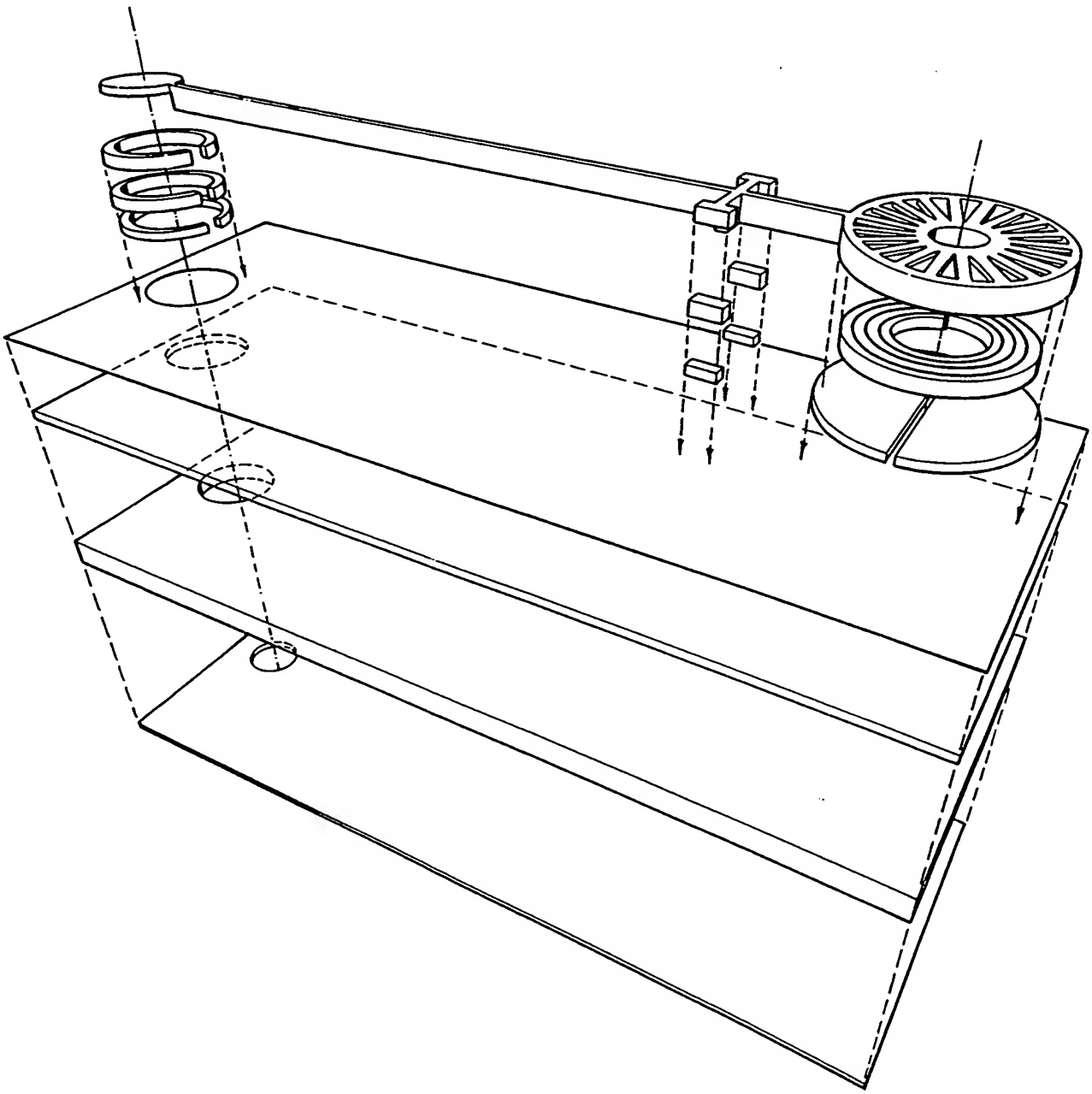


Fig. C05.2

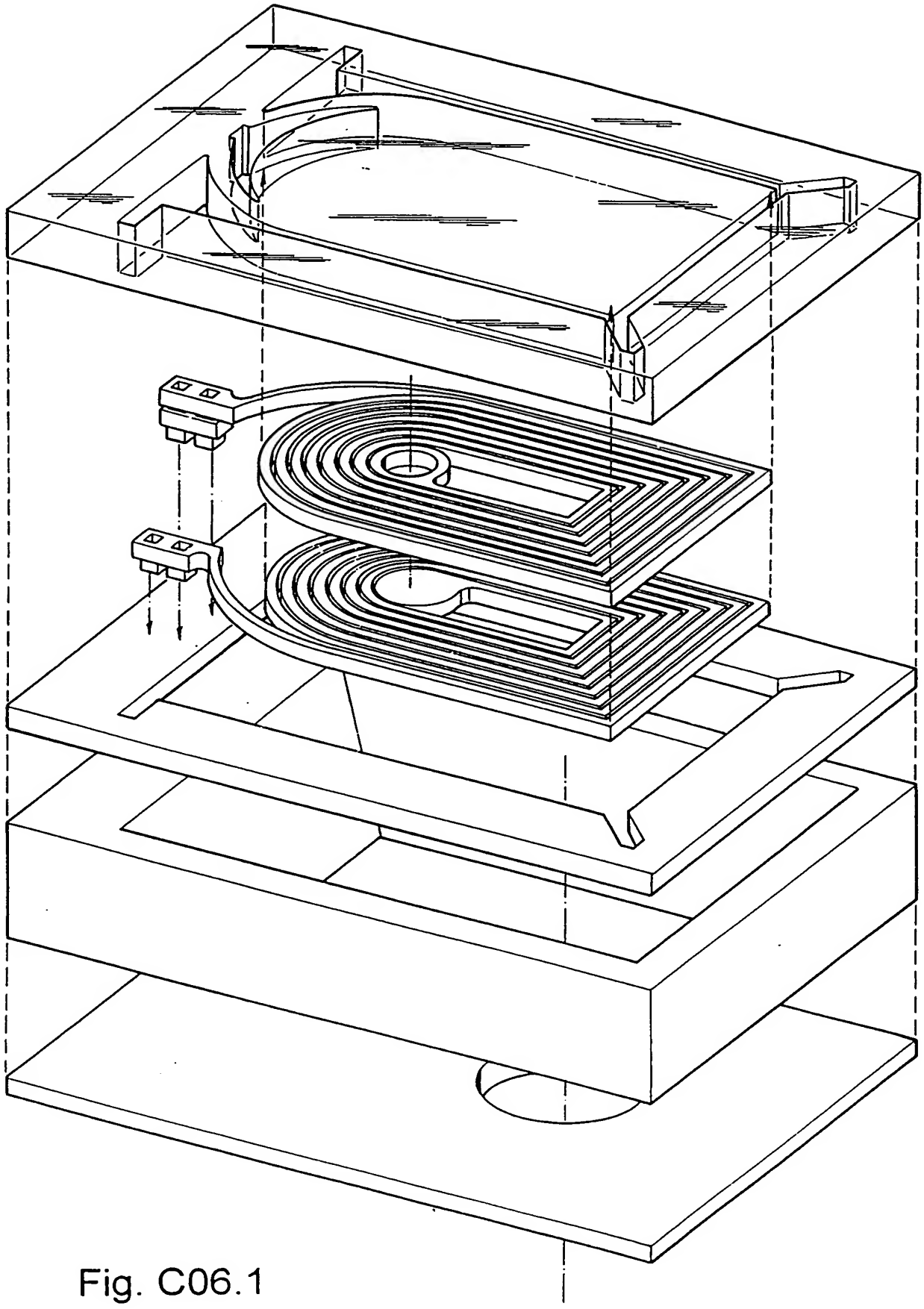


Fig. C06.1

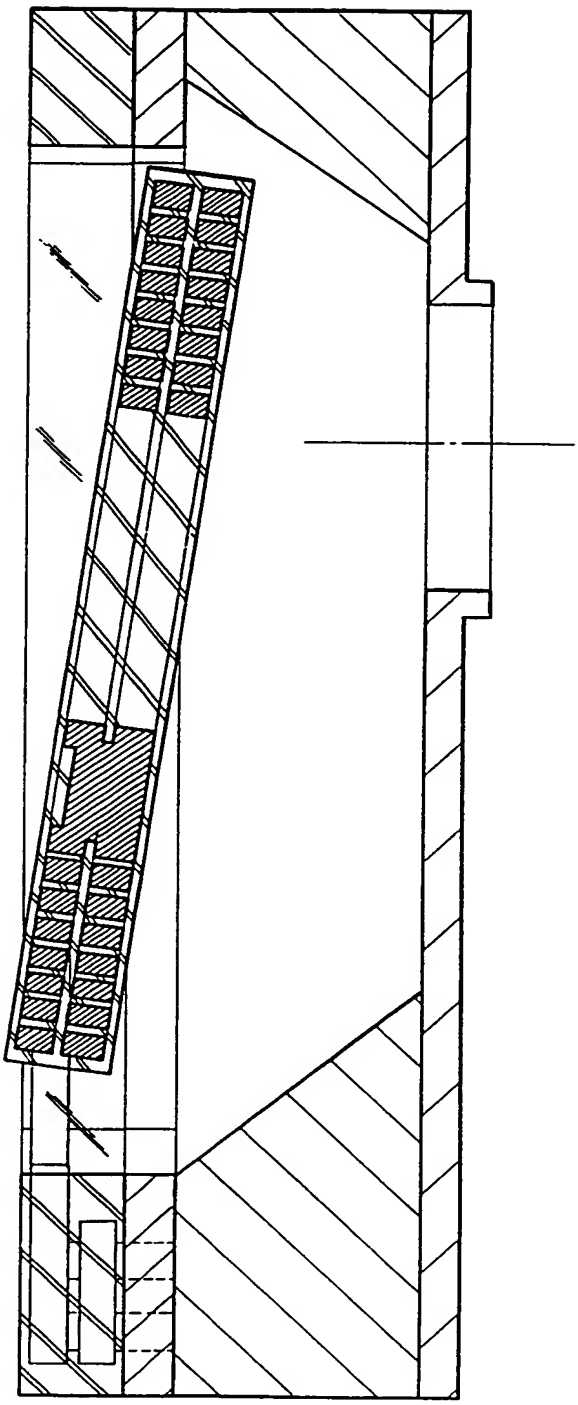


Fig. C06.2

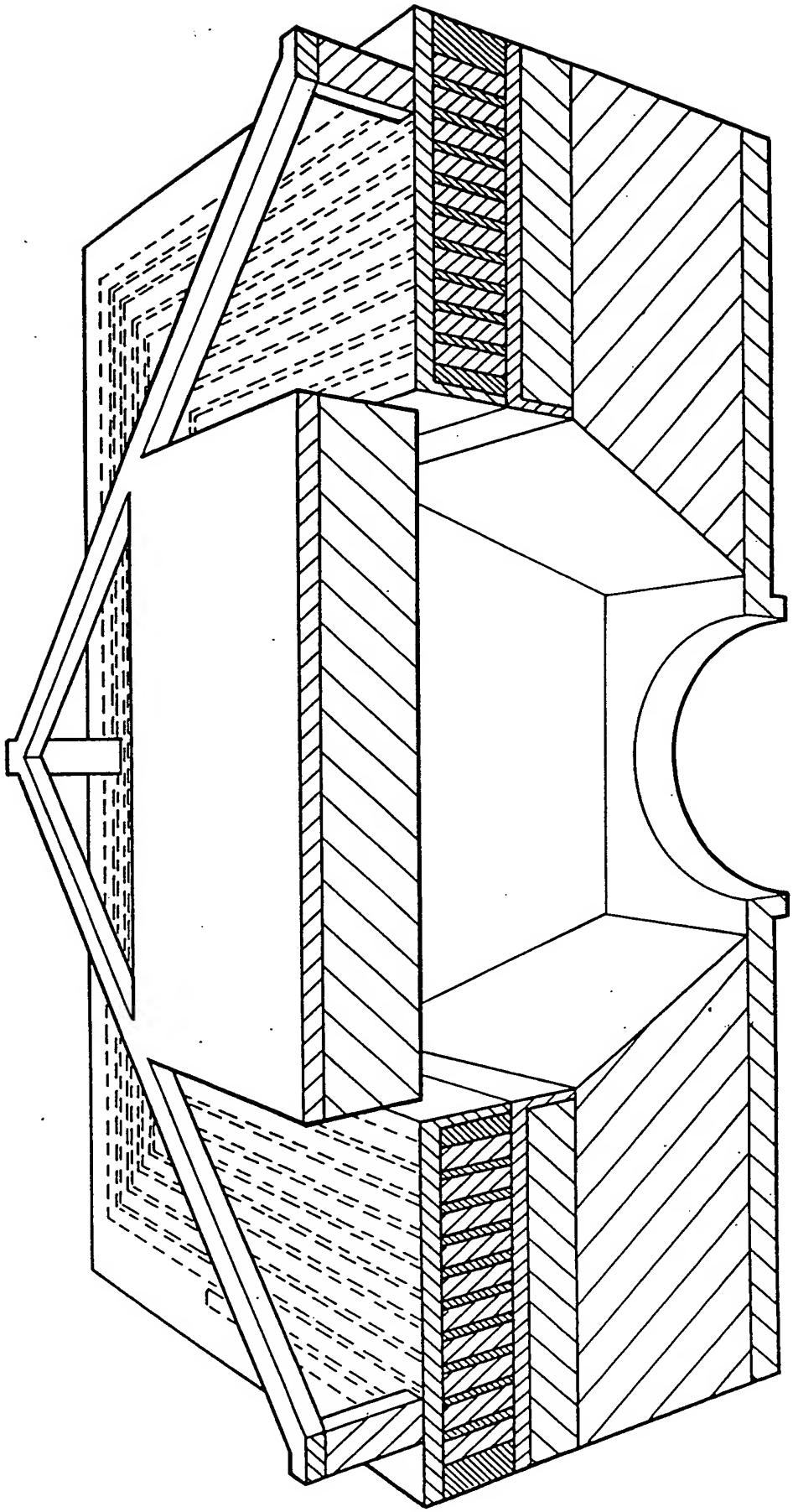


Fig. C07.1

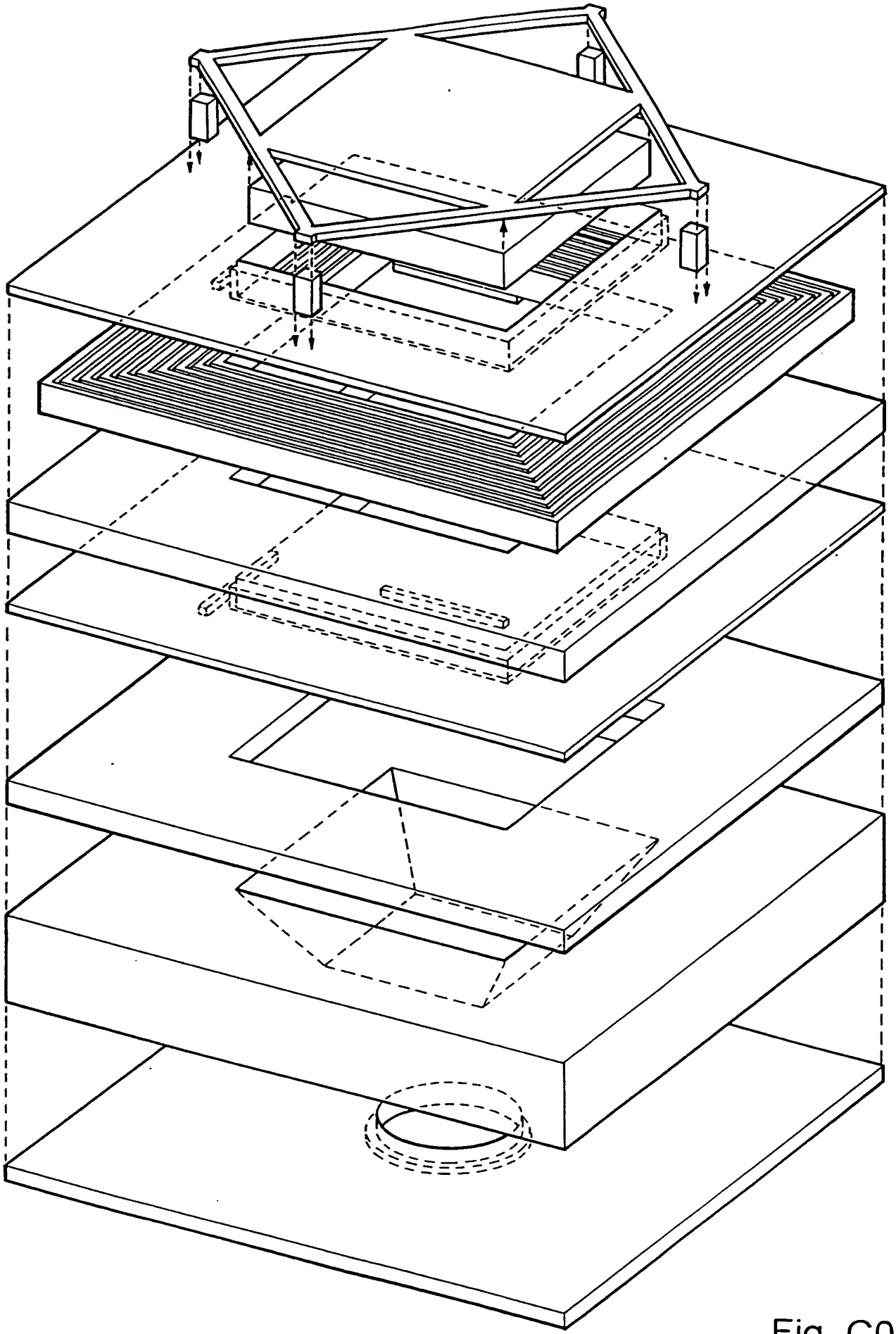


Fig. C07.2

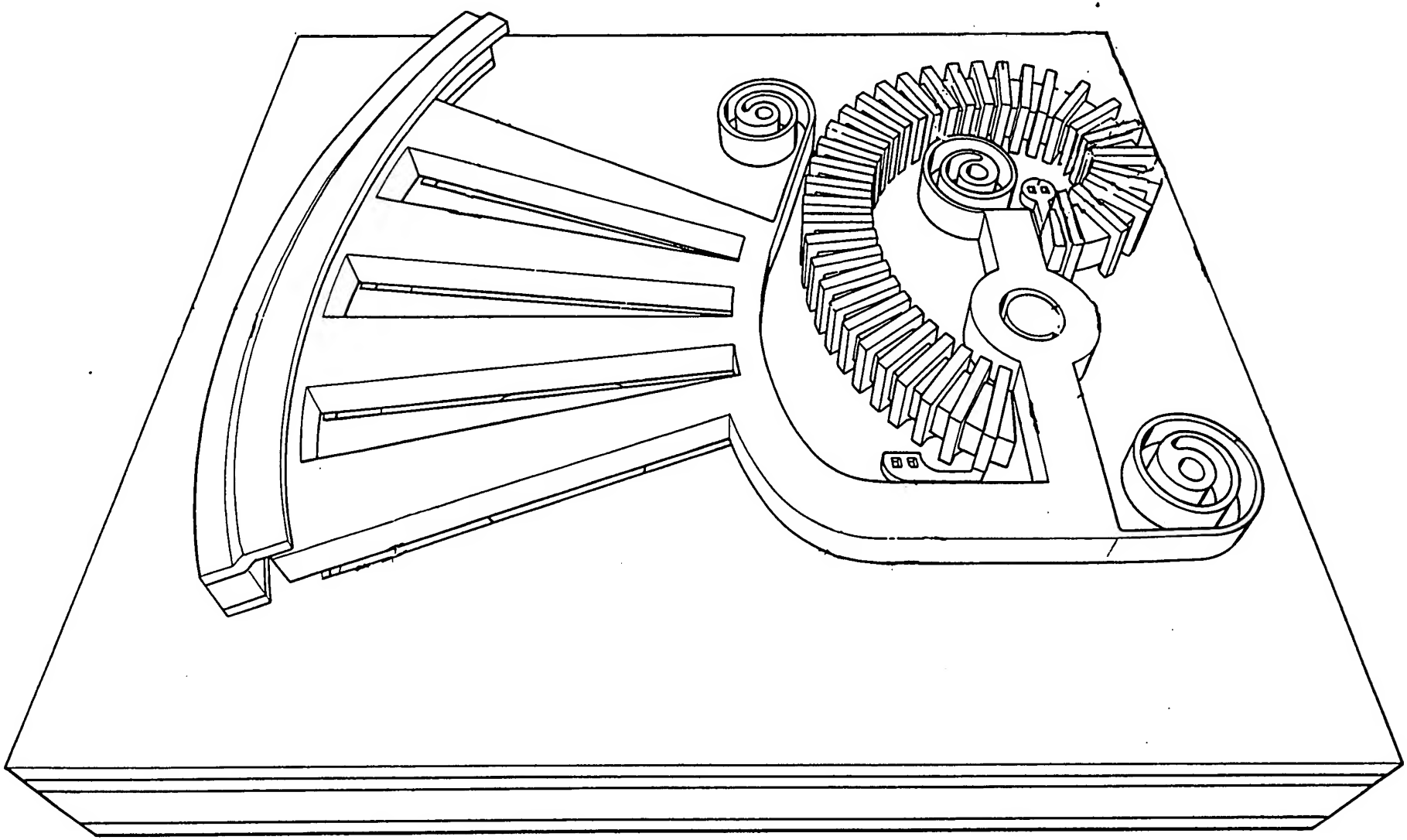


Fig. C08.1

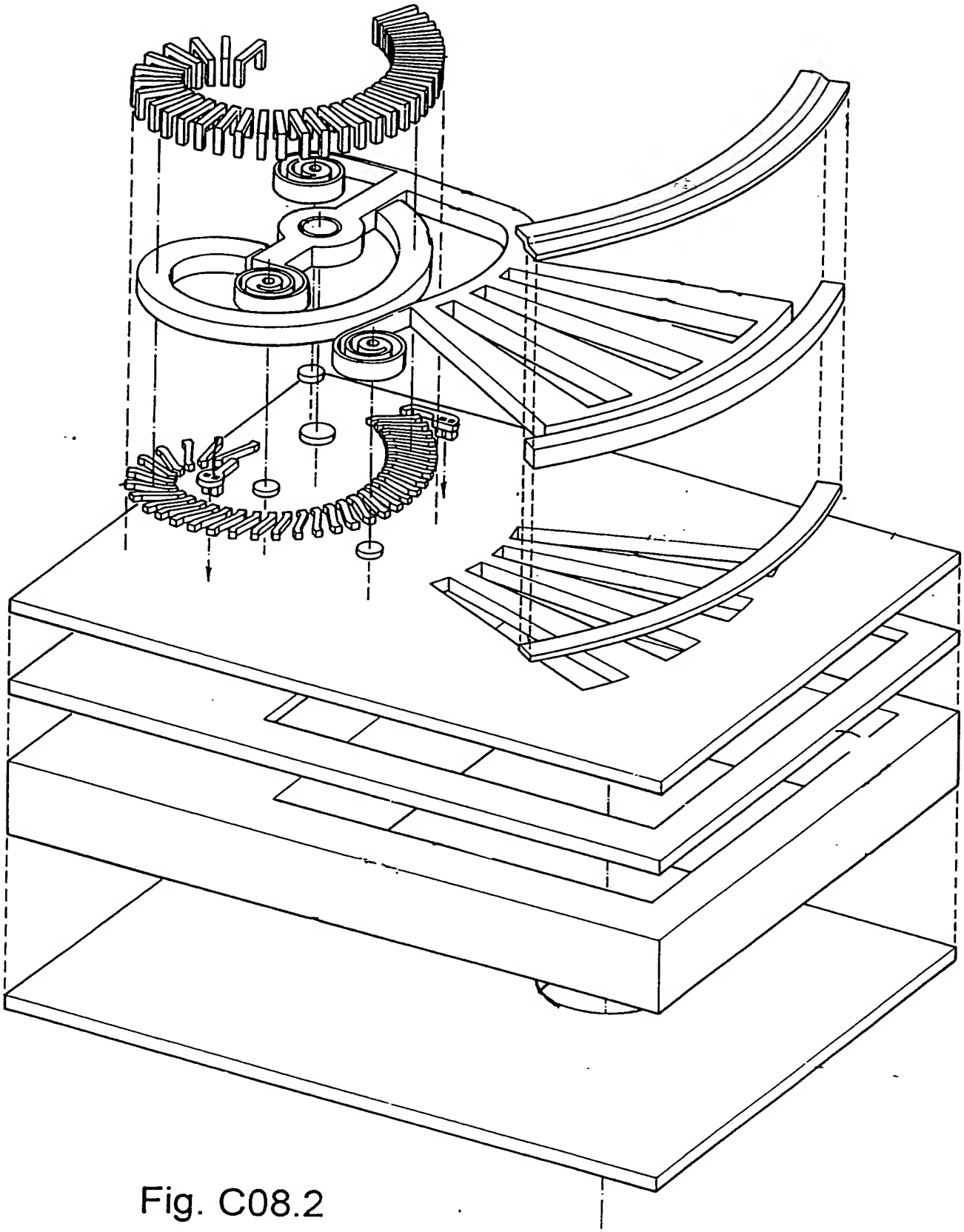


Fig. C08.2

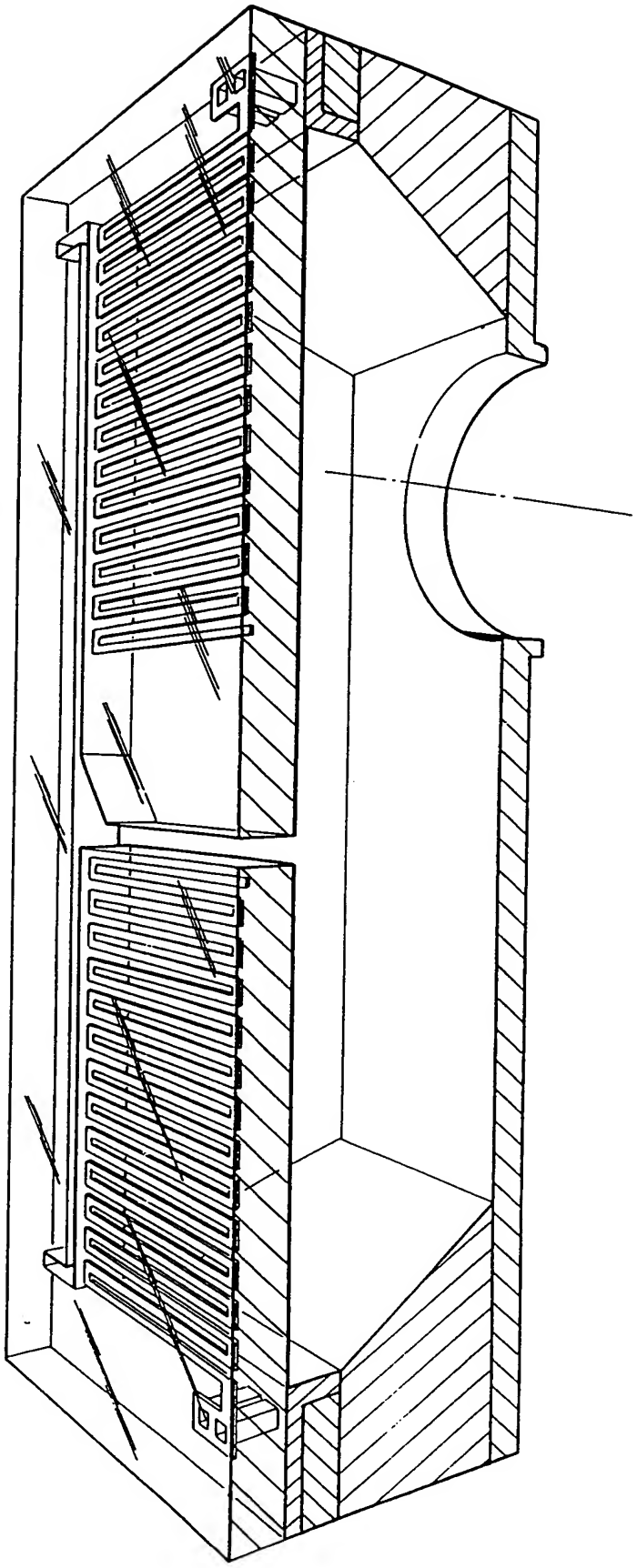


Fig. C09.1

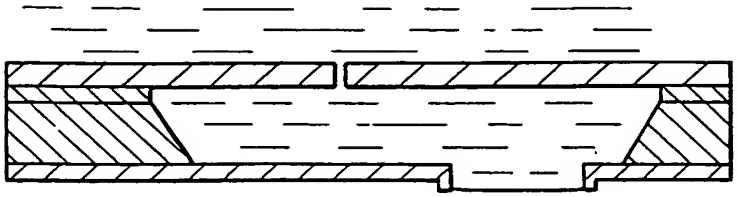


Fig. C09.2

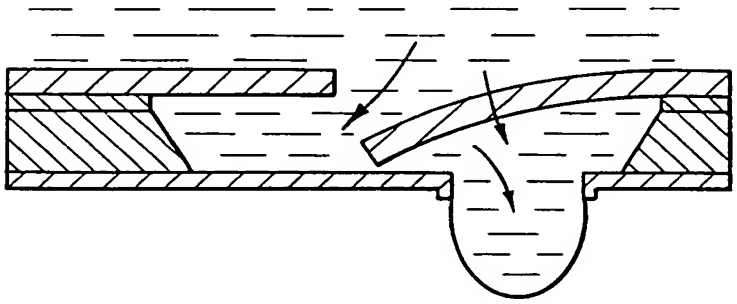


Fig. C09.3

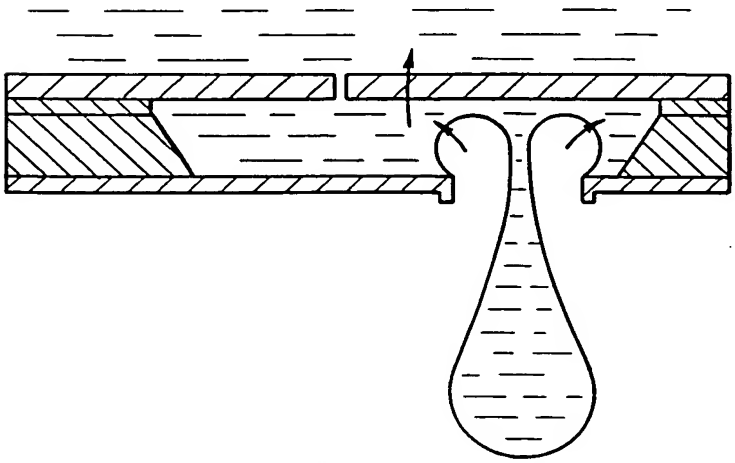


Fig. C09.4

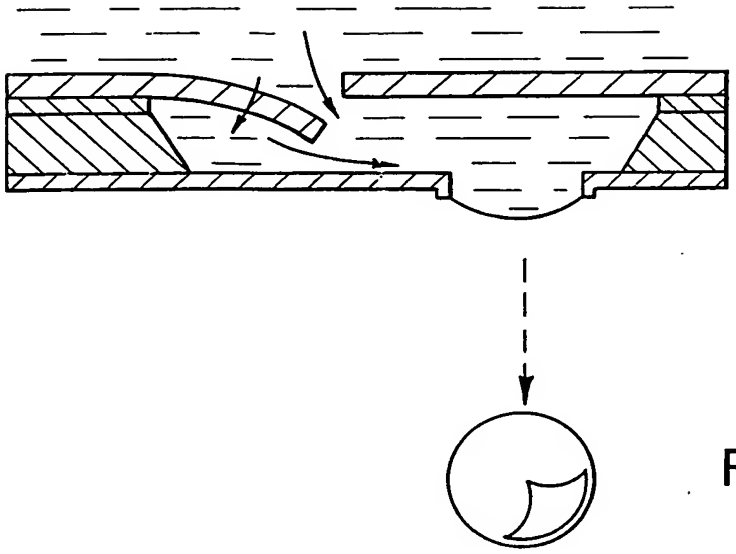


Fig. C09.5

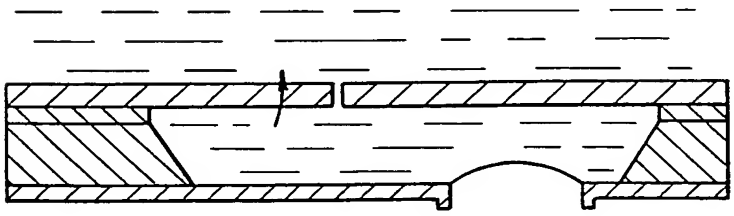


Fig. C09.6

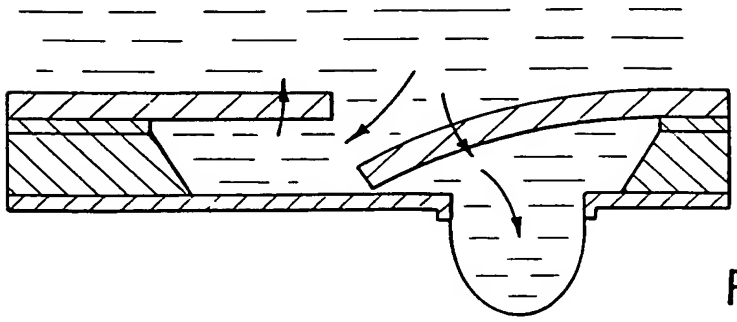


Fig. C09.7

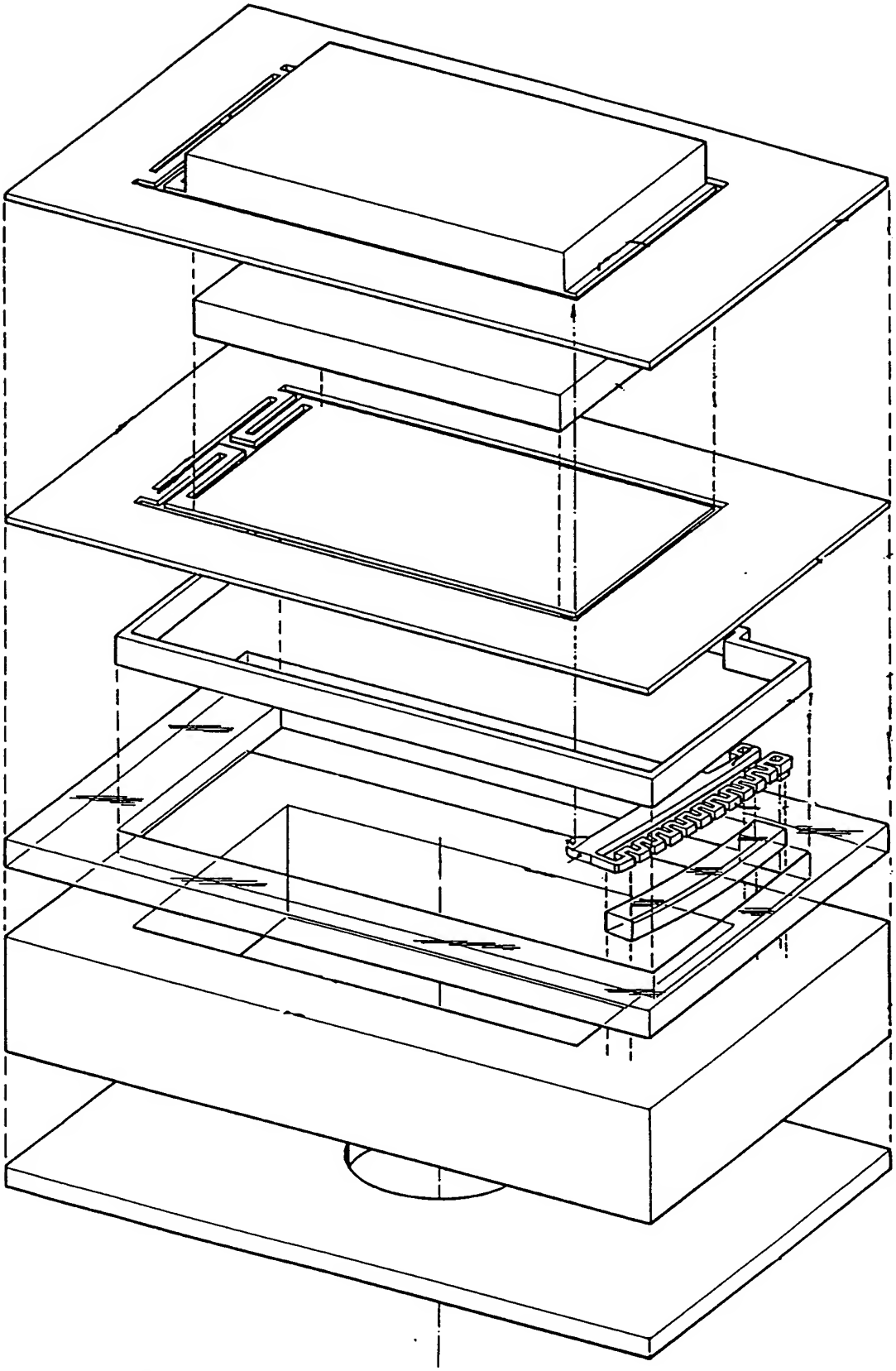


Fig. C10.1

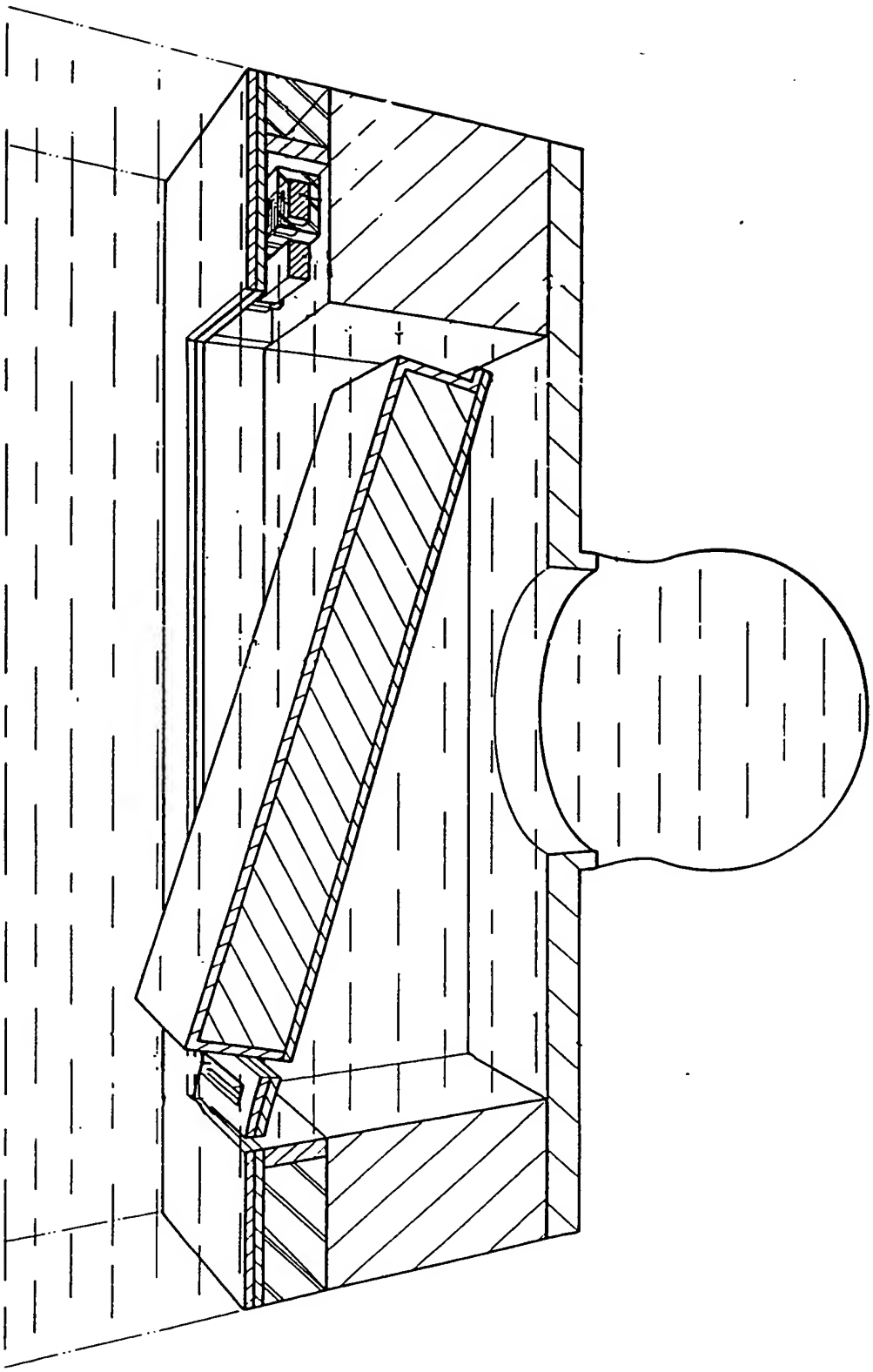


Fig. C10.2

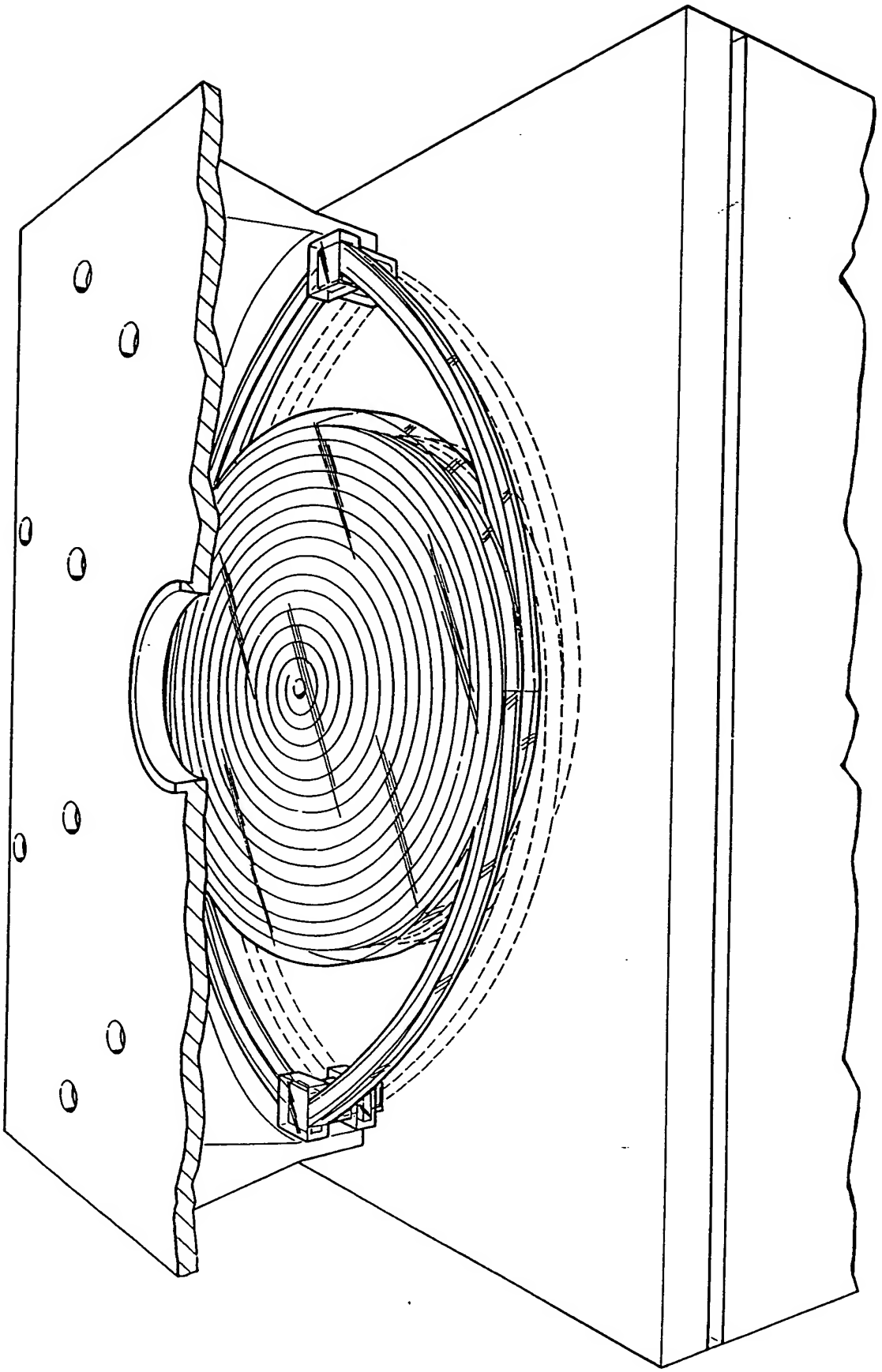


Fig. C11.1

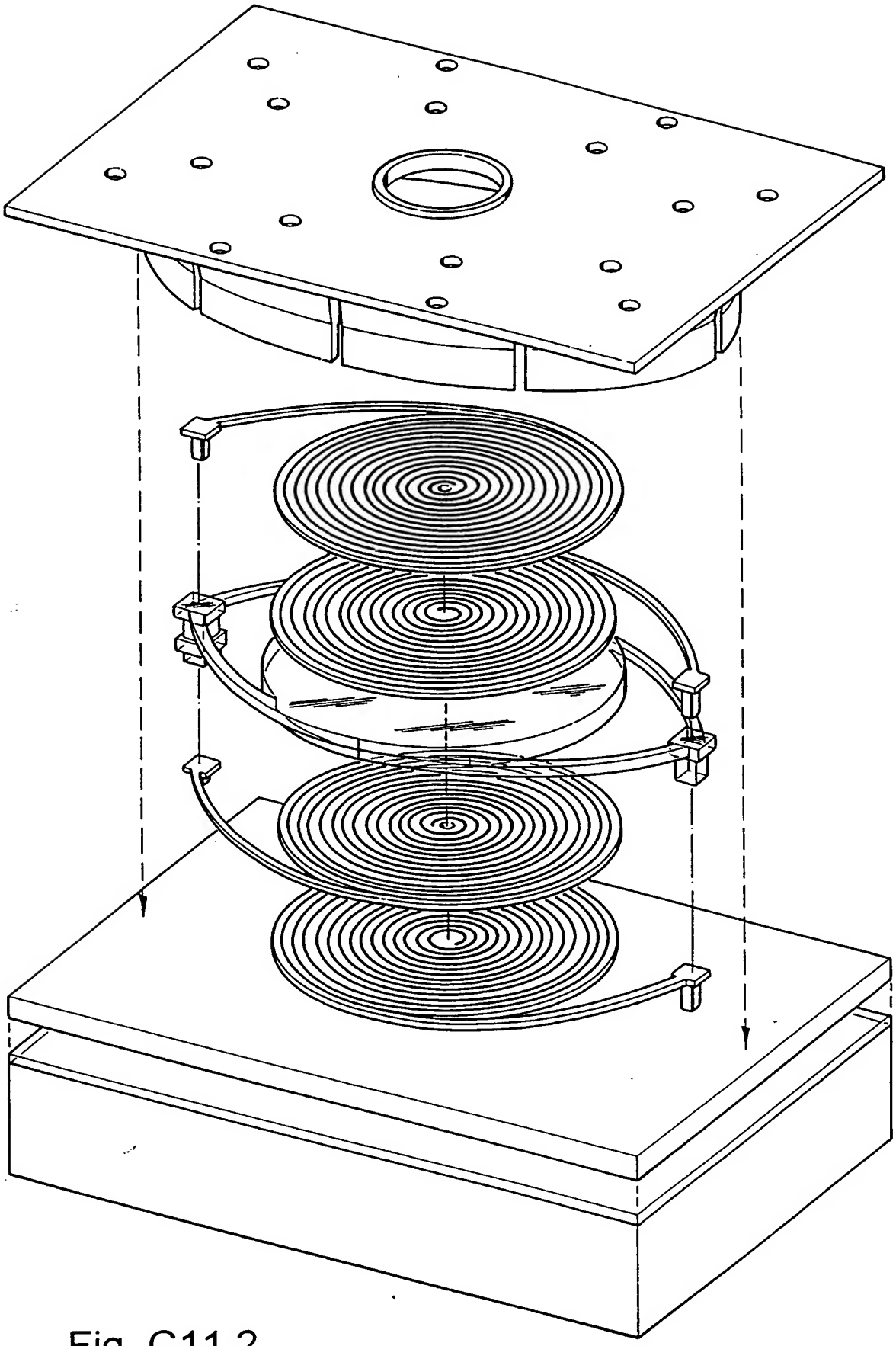


Fig. C11.2

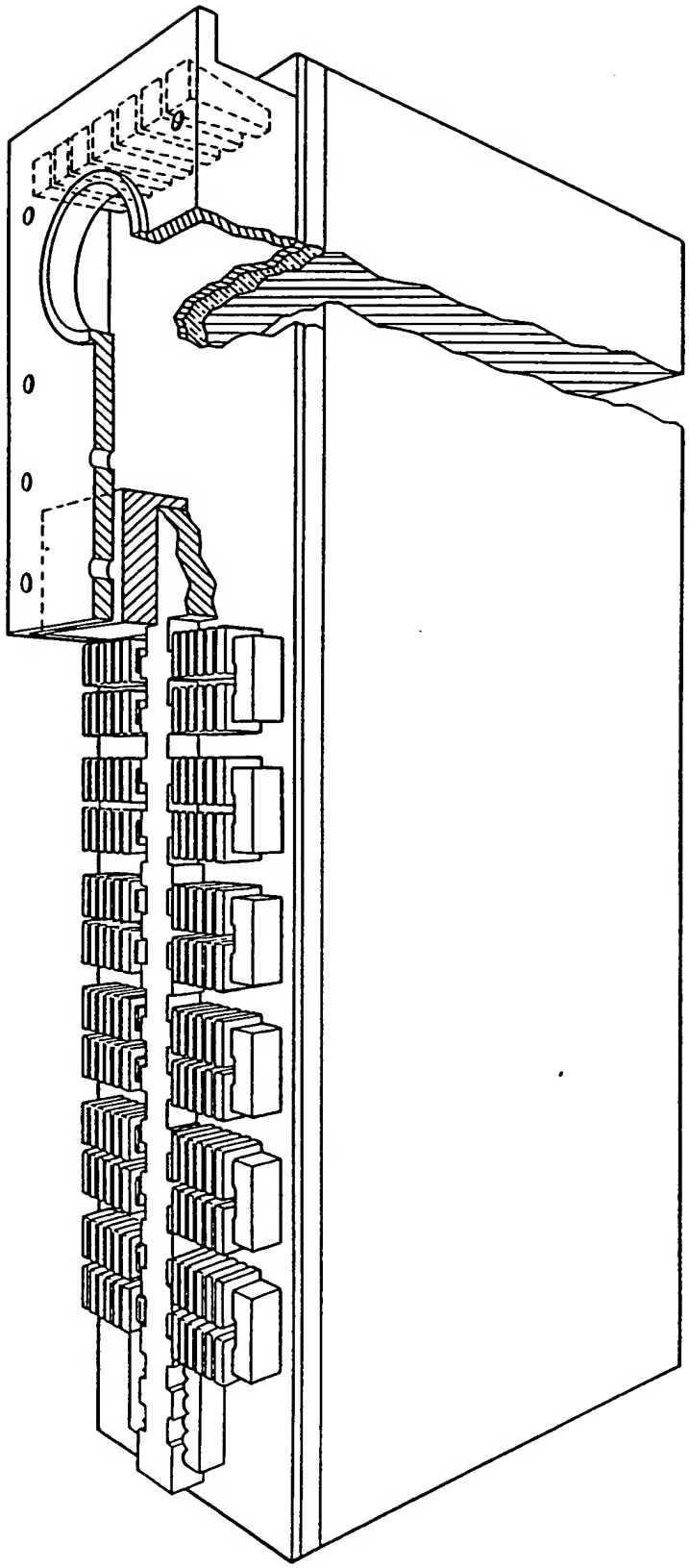


Fig. C12.1

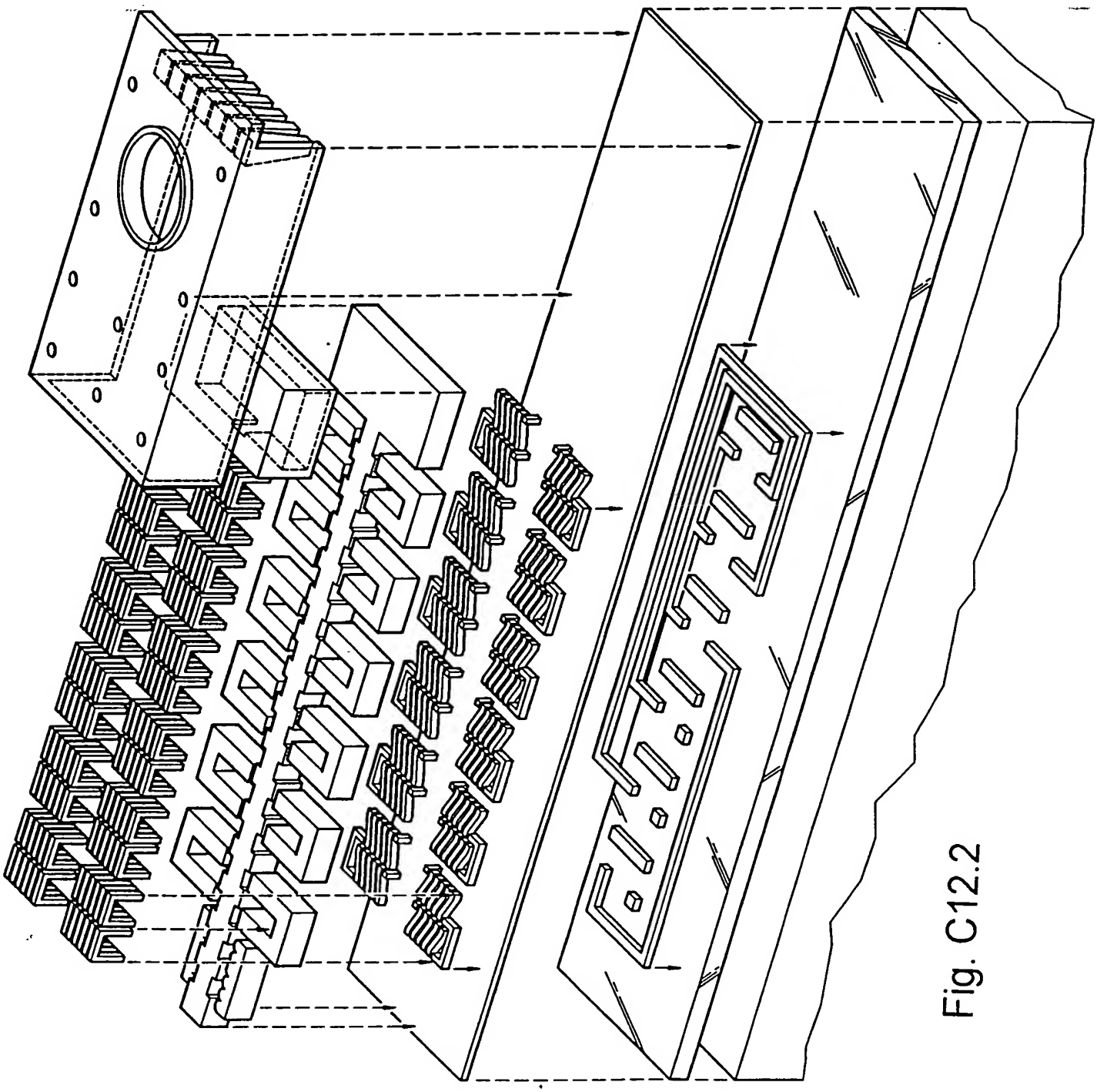


Fig. C12.2

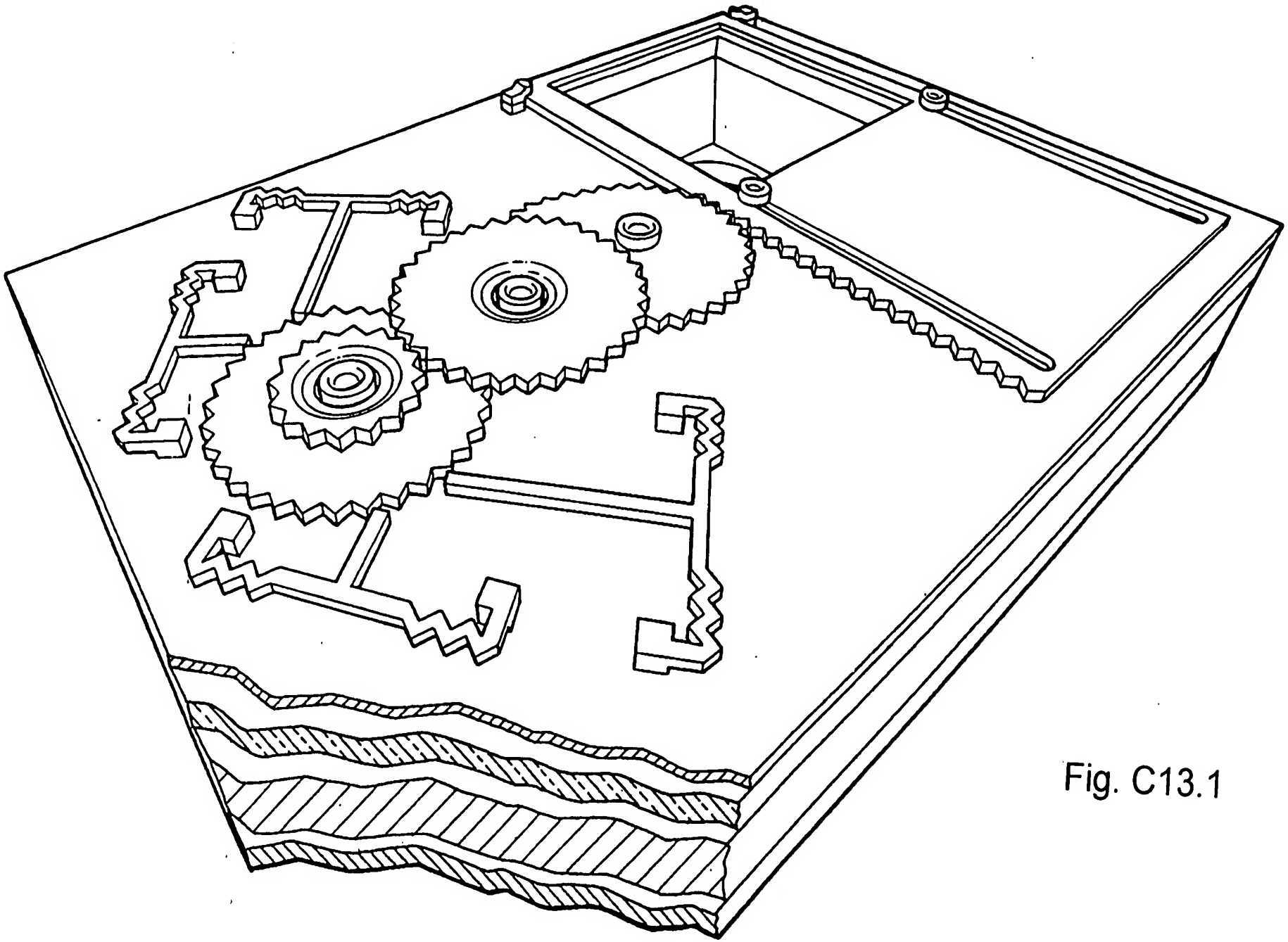


Fig. C13.1

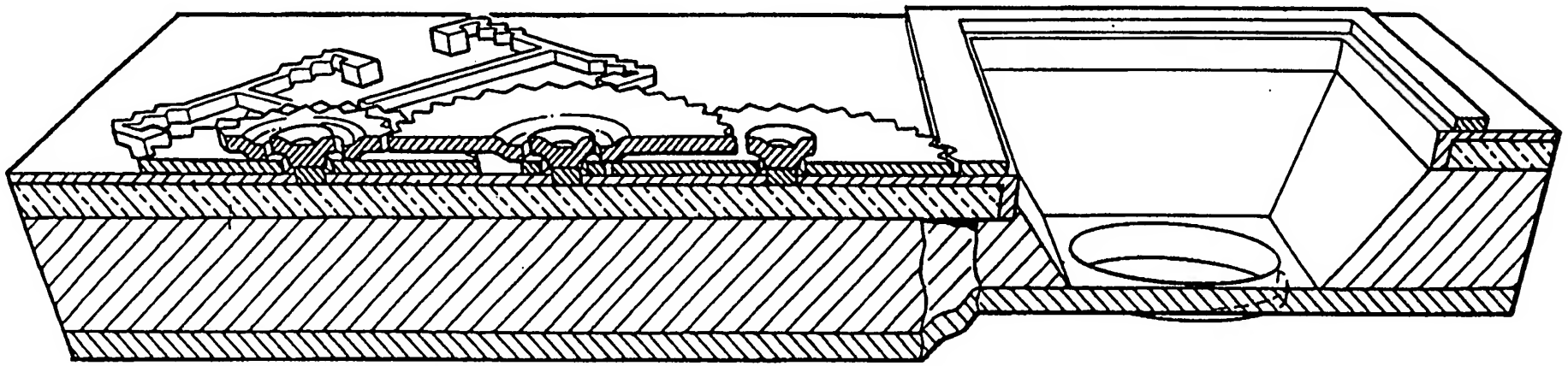


Fig. C13.2

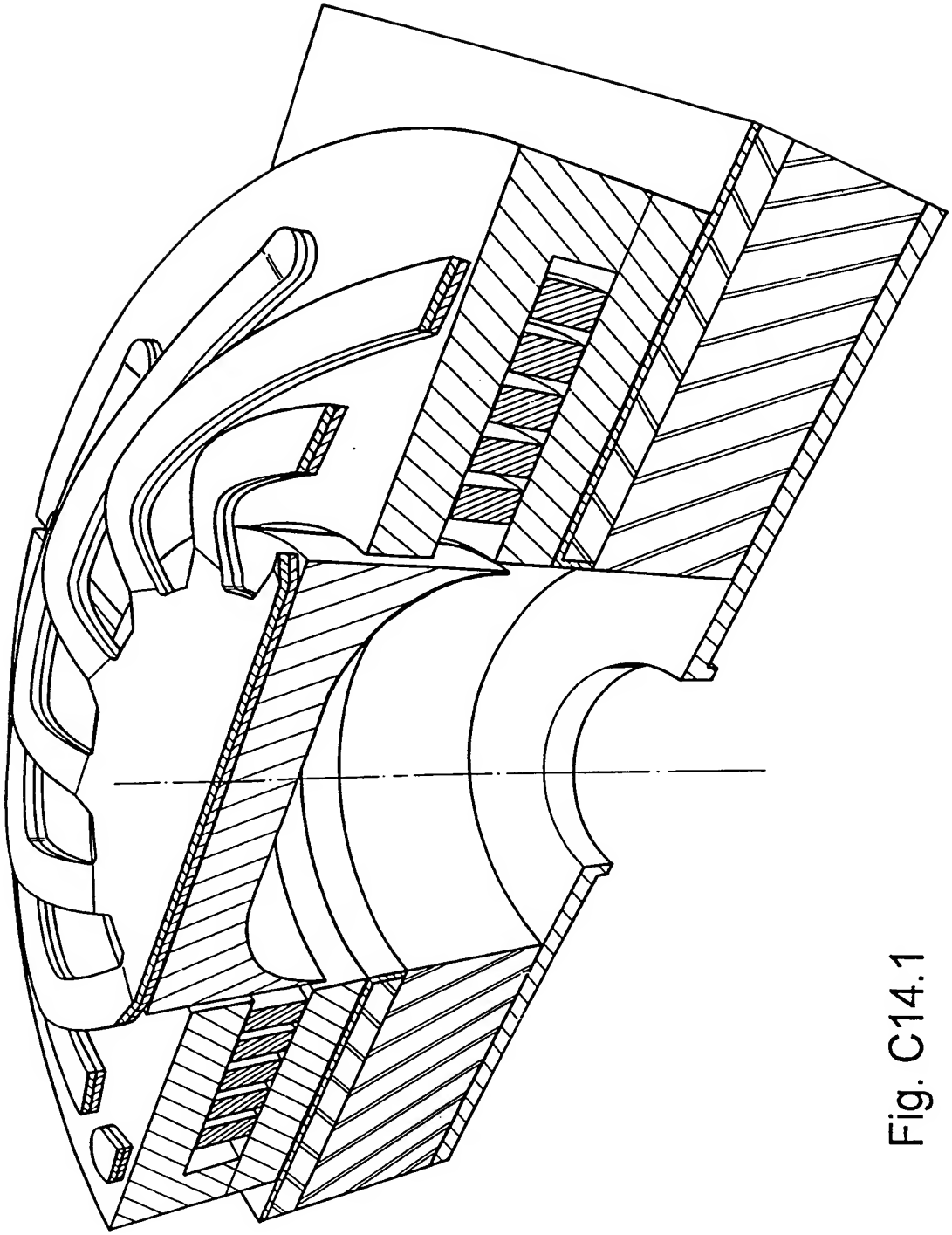


Fig. C14.1

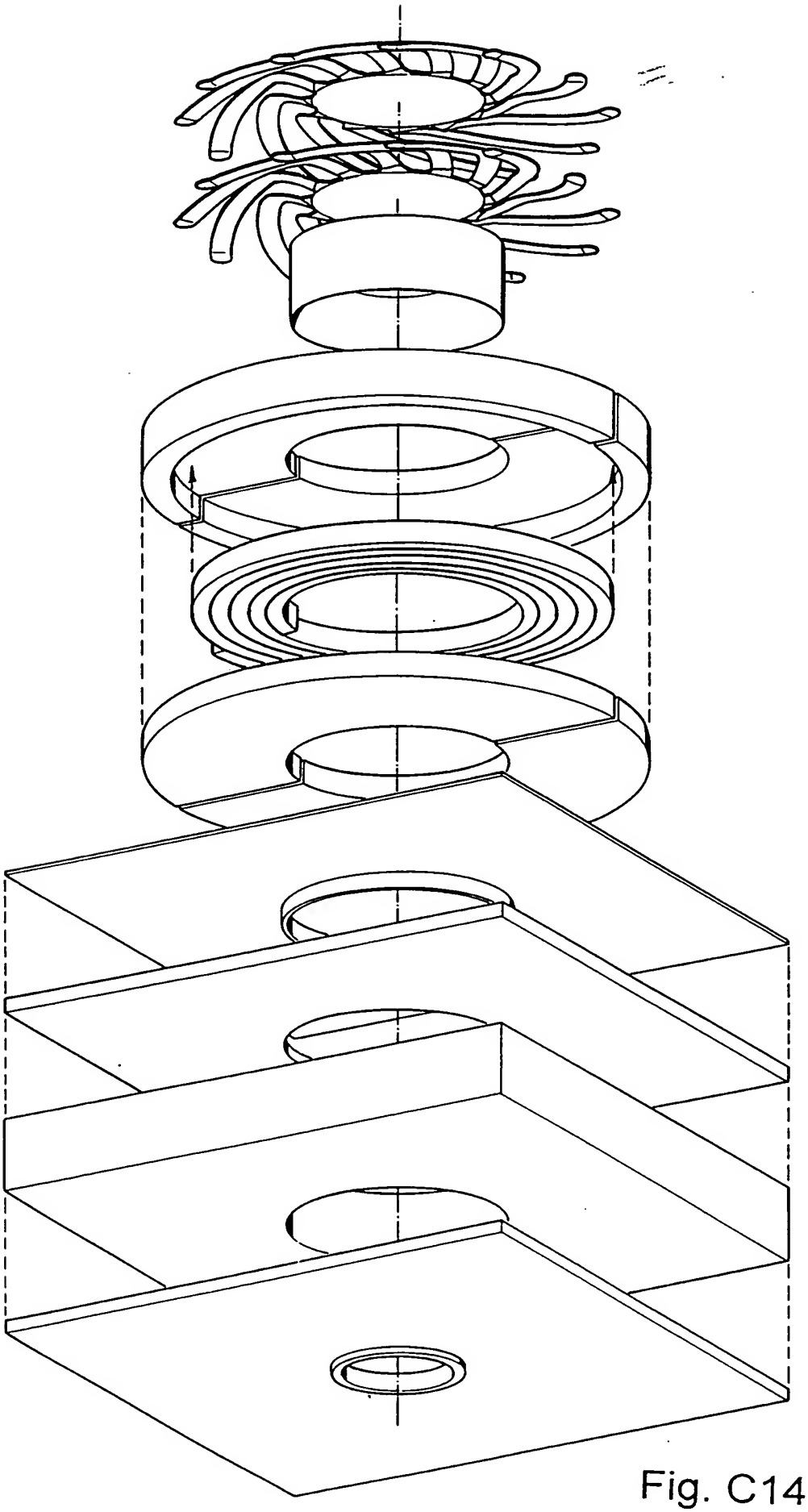


Fig. C14.2

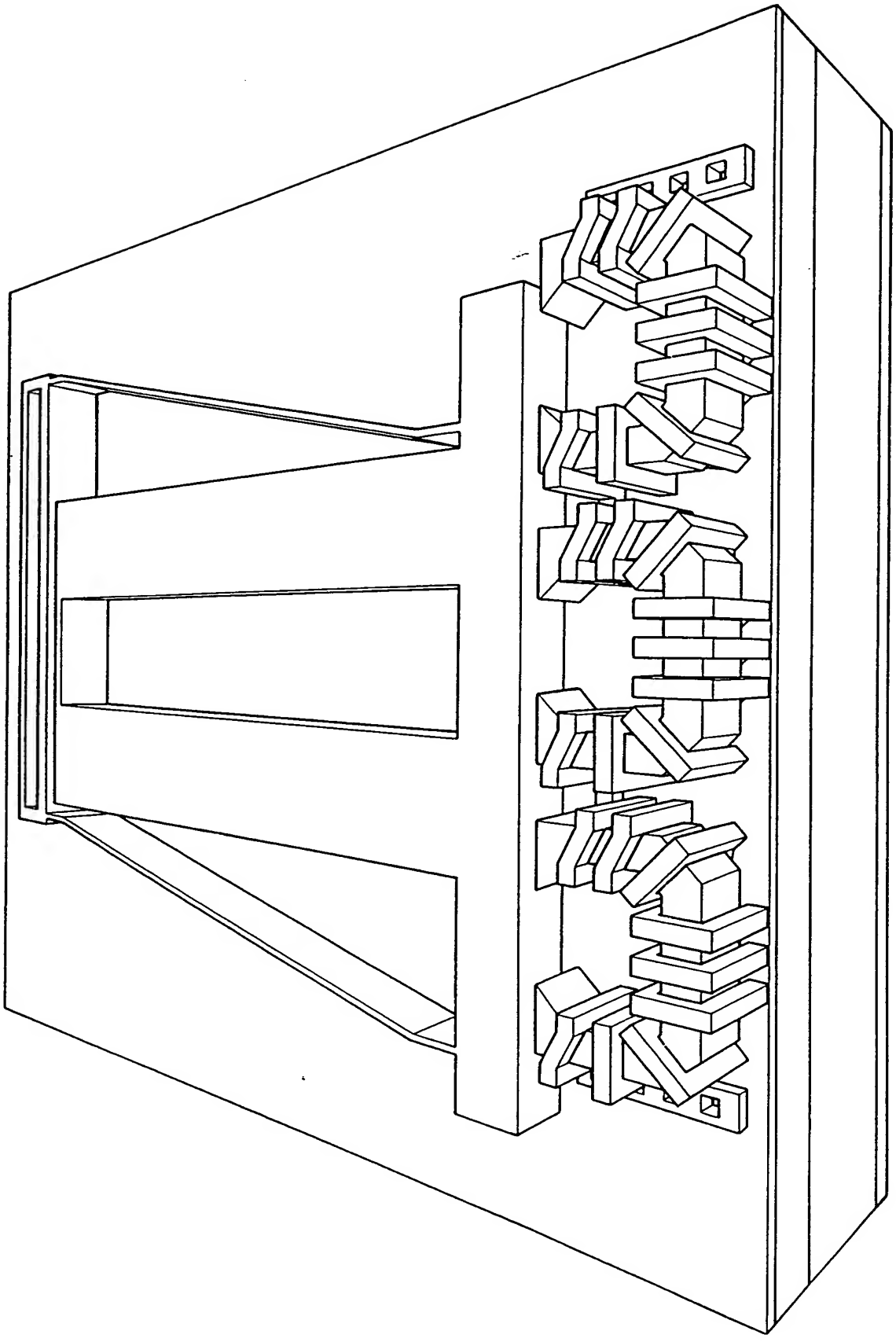


Fig. C15.1

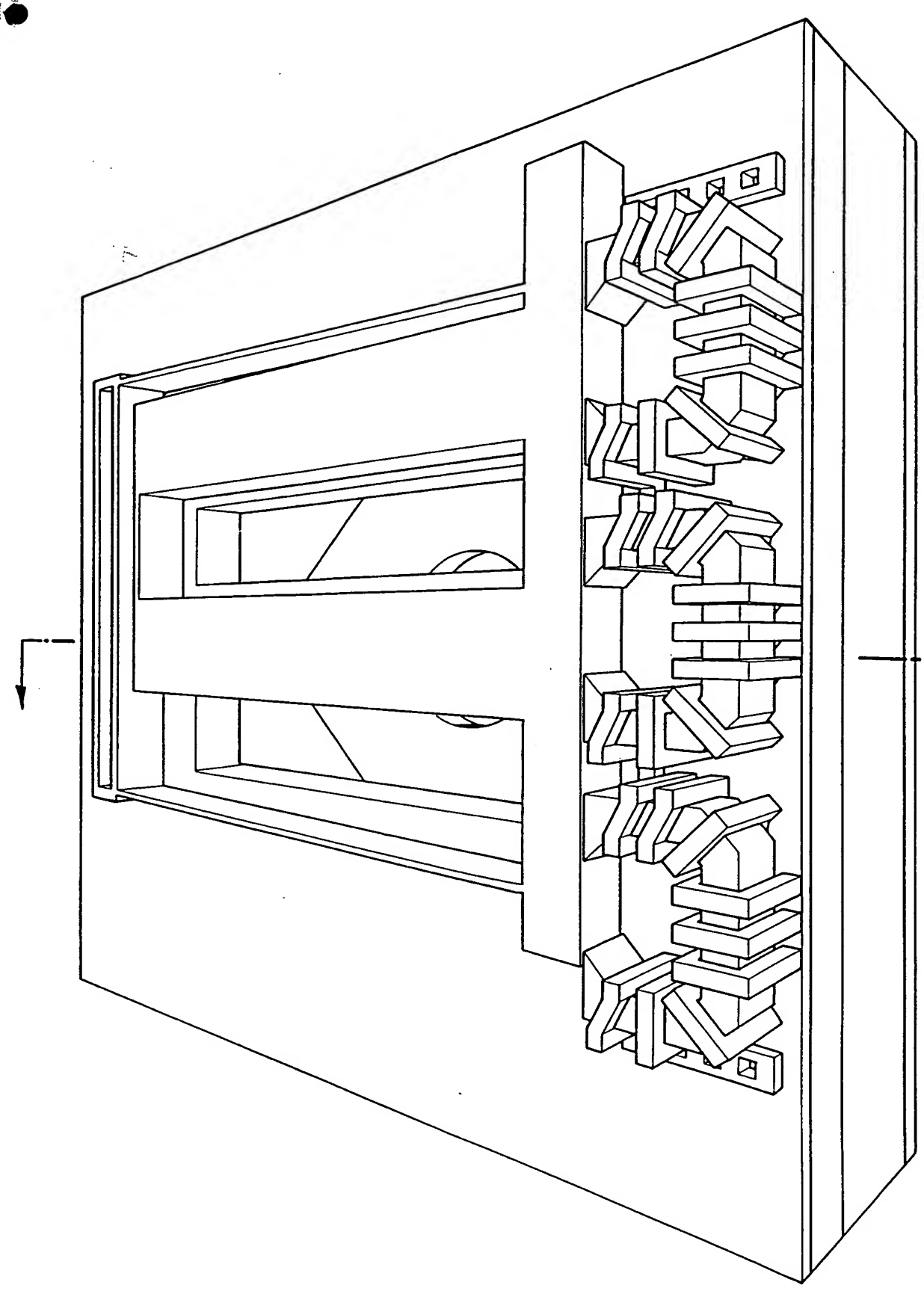


Fig. C15.2

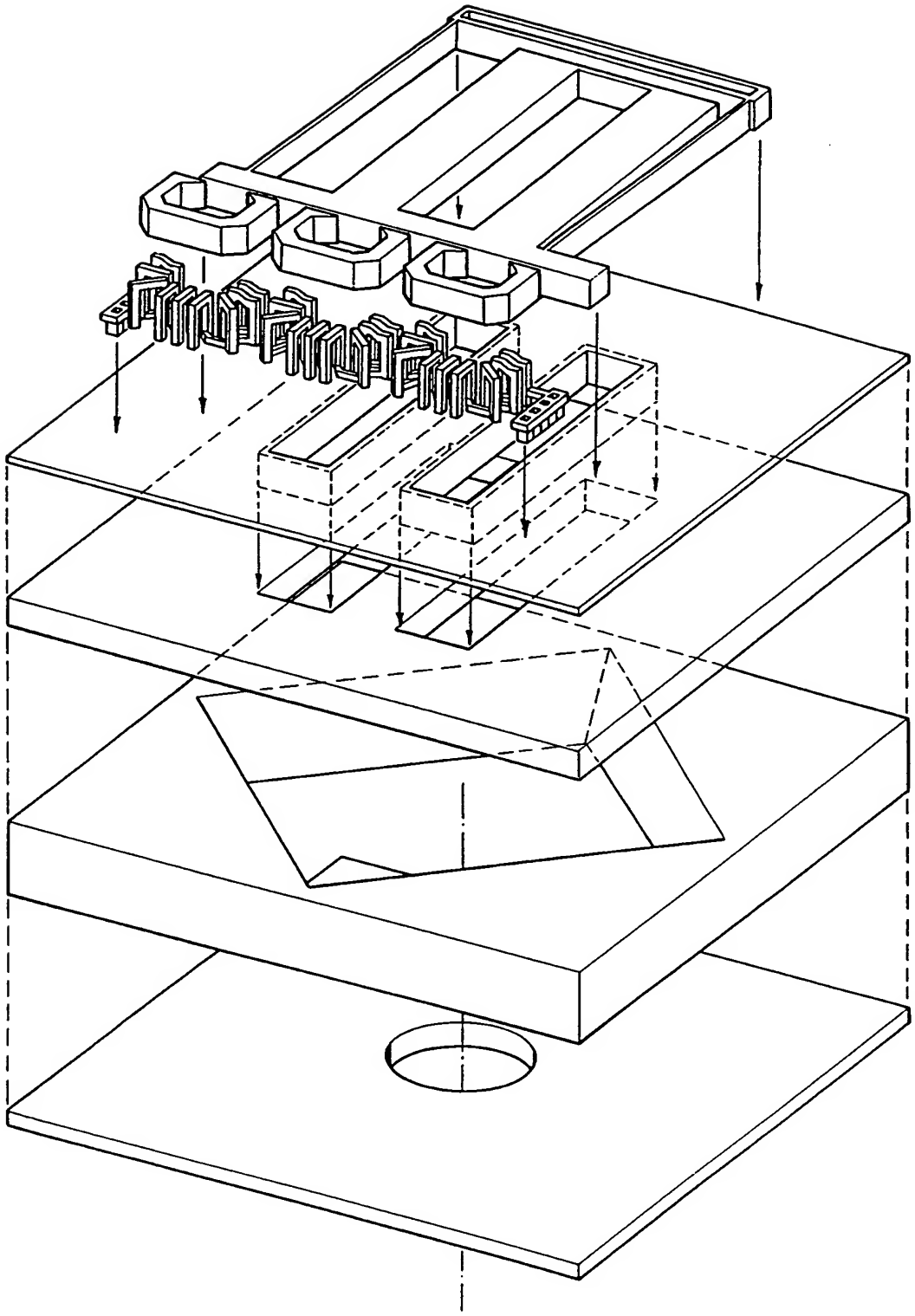


Fig. C15.3

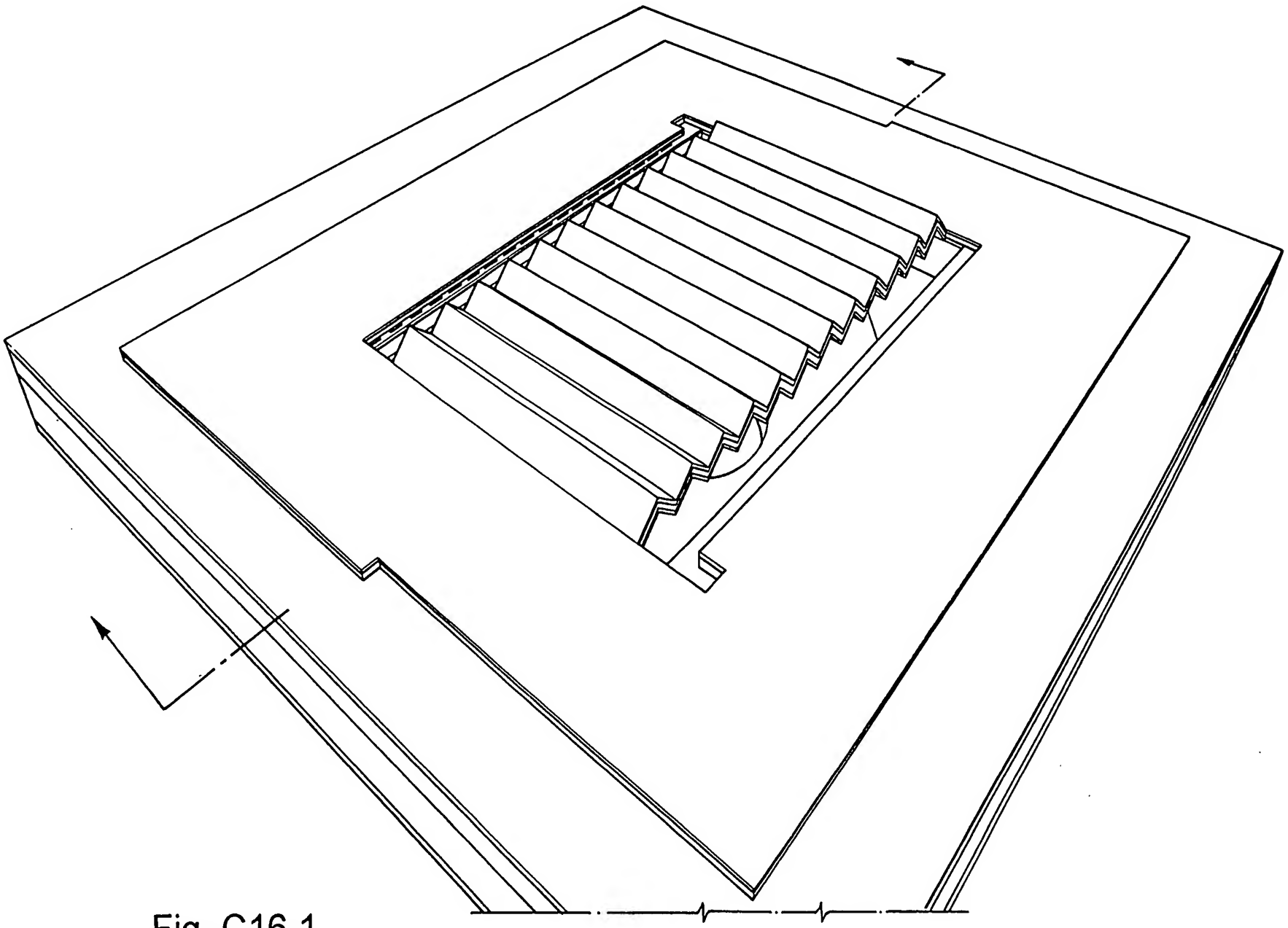


Fig. C16.1

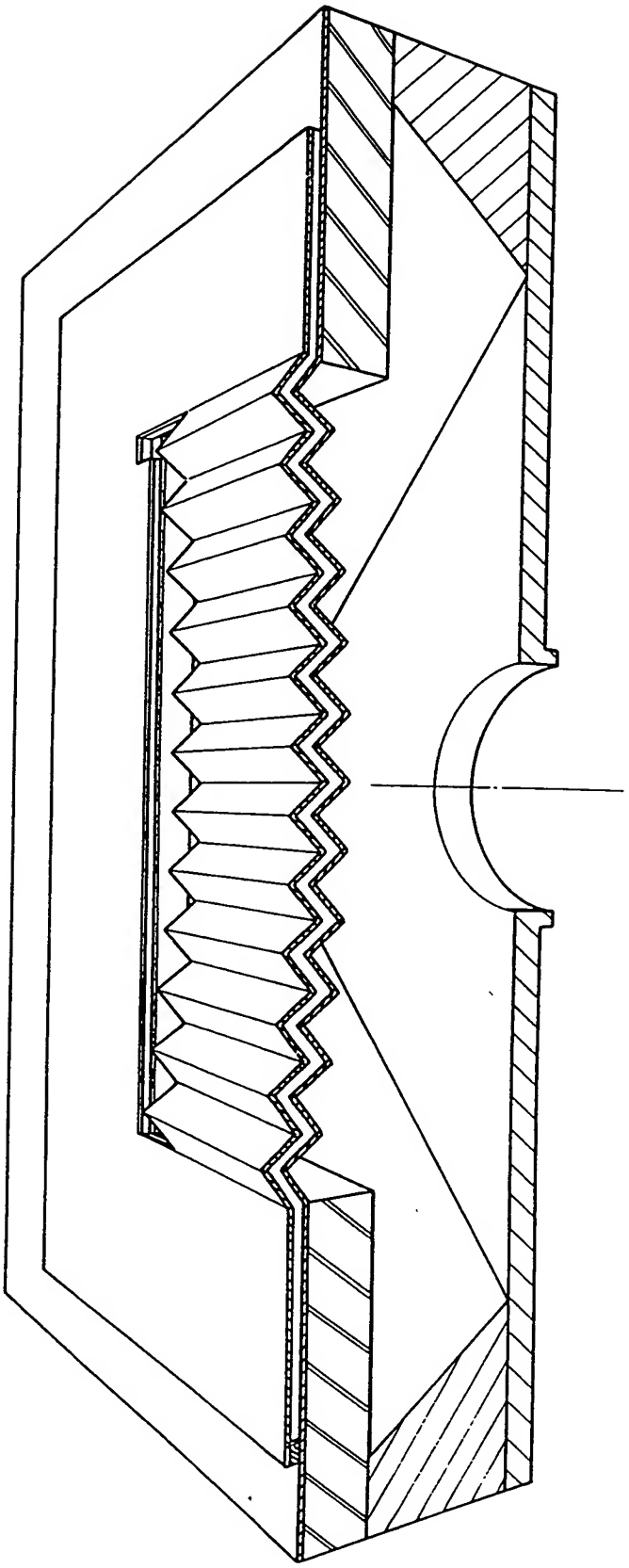


Fig. C16.2

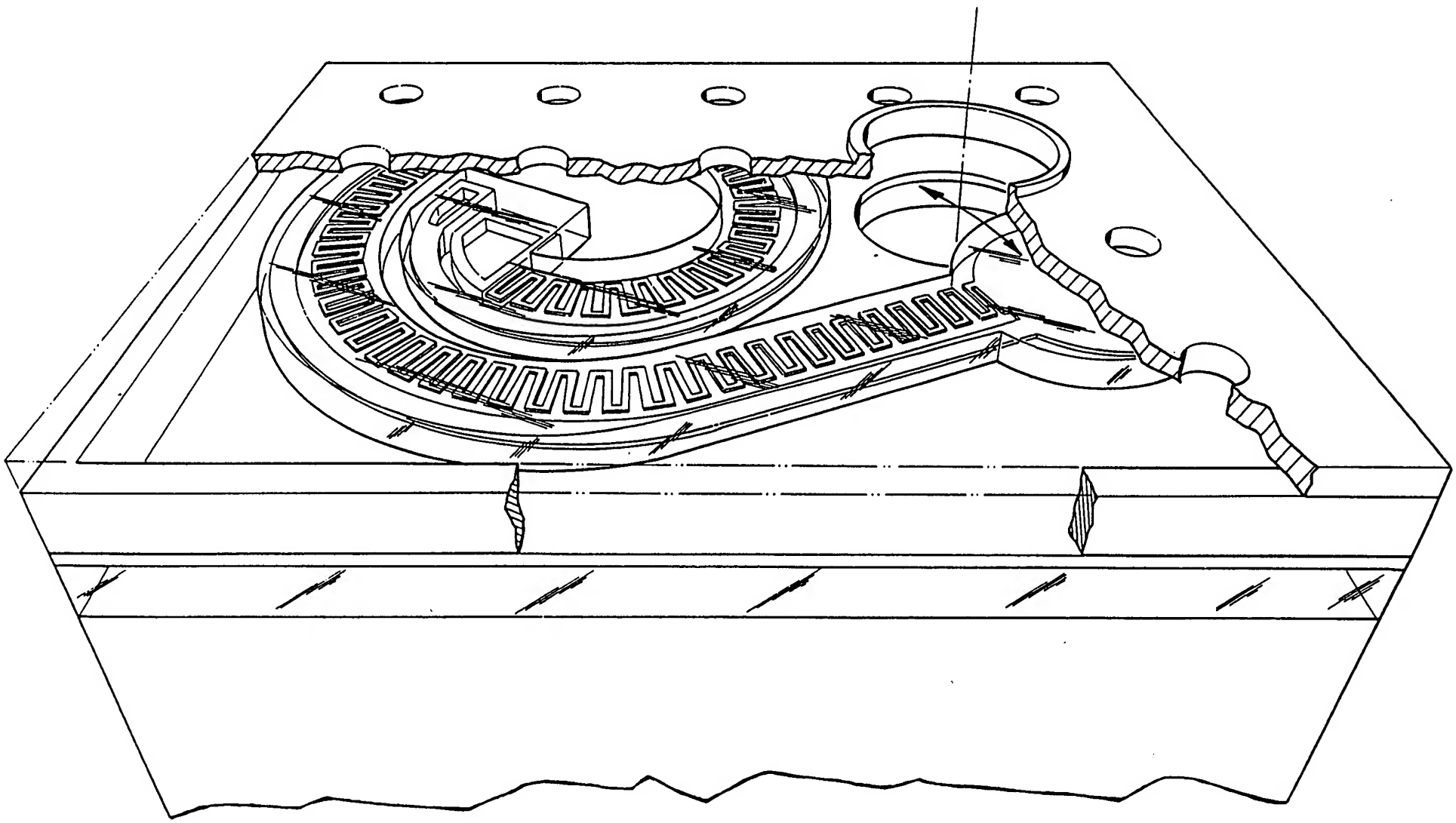


Fig. C17.1

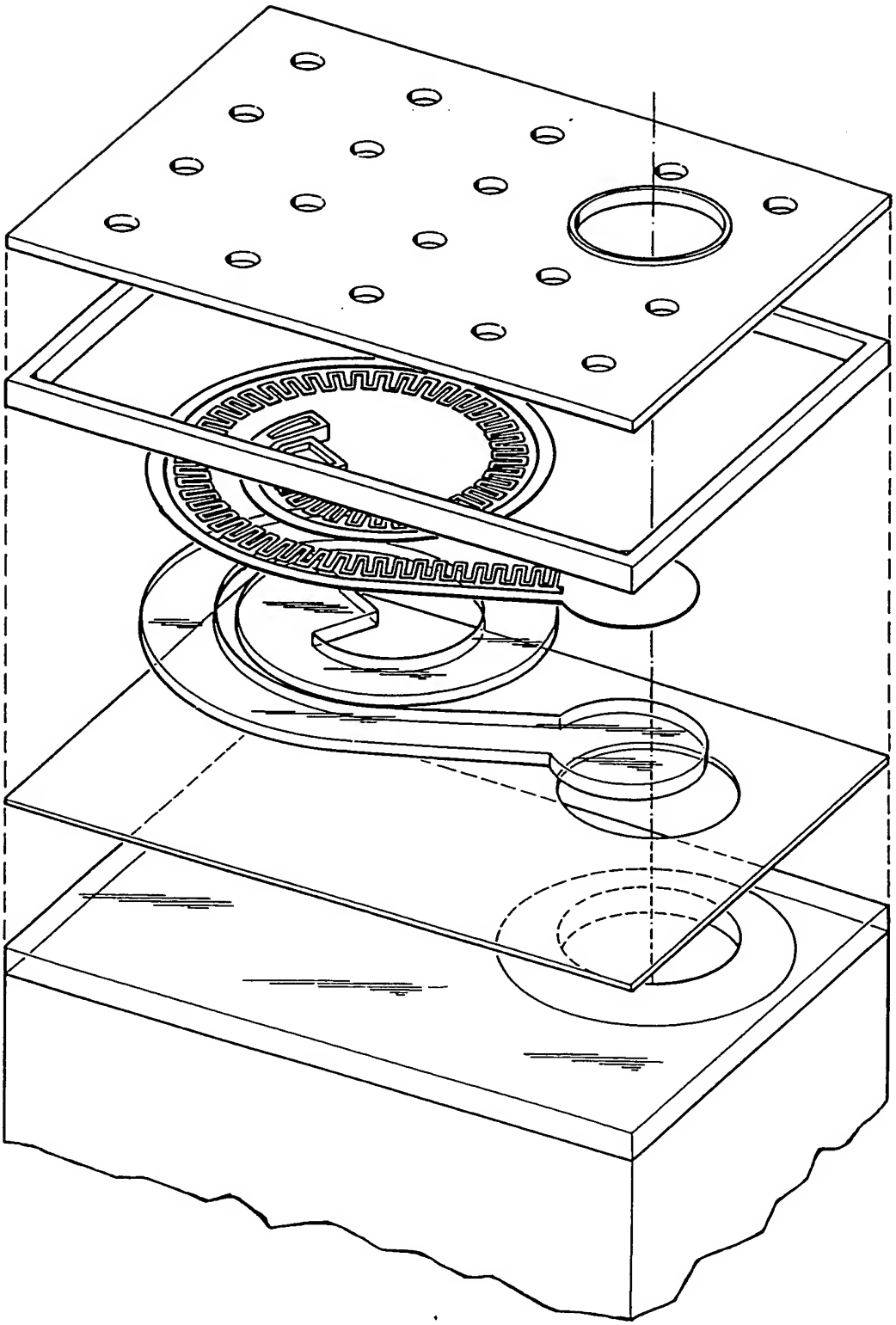


Fig. C17.2

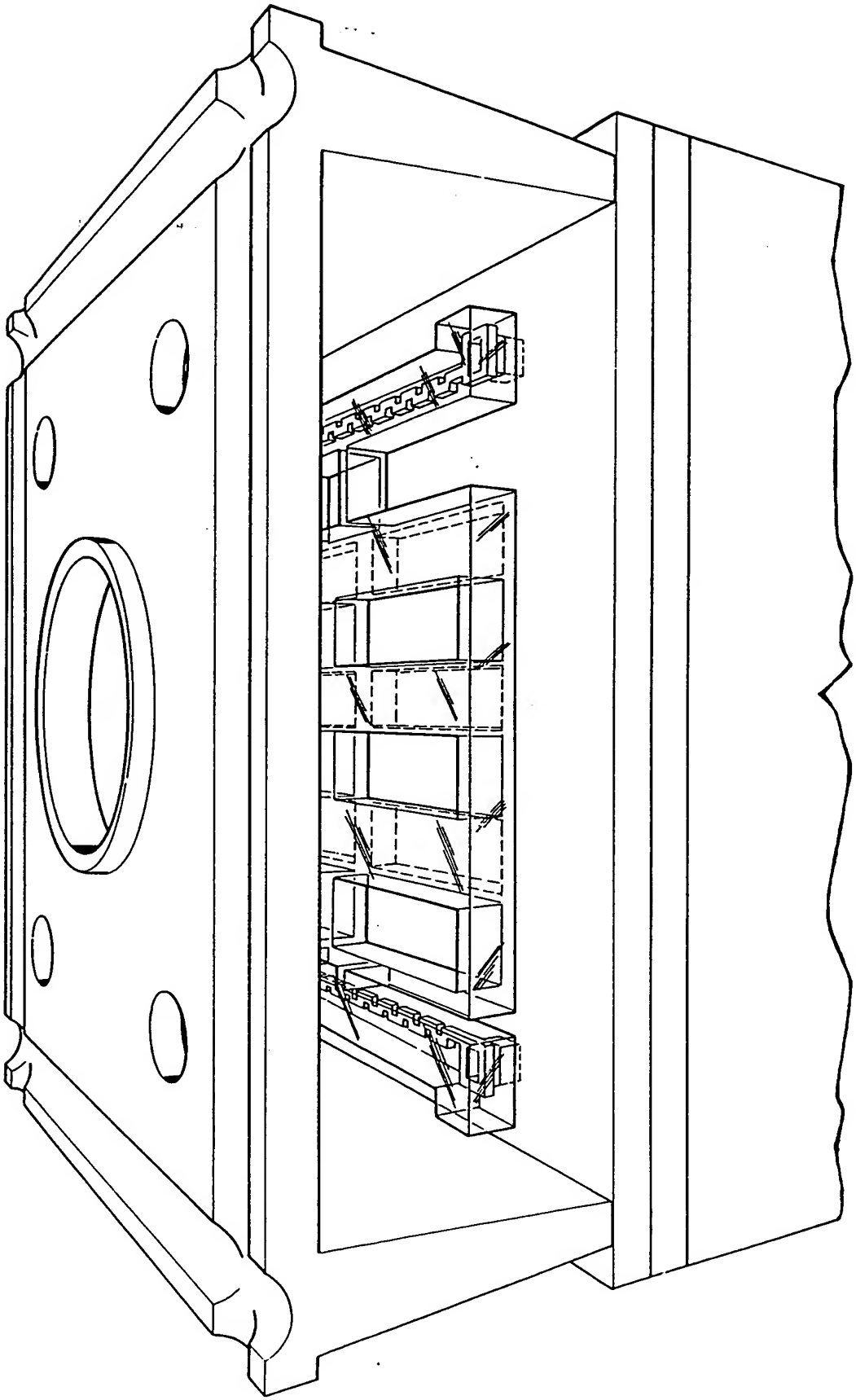


Fig. C18.1

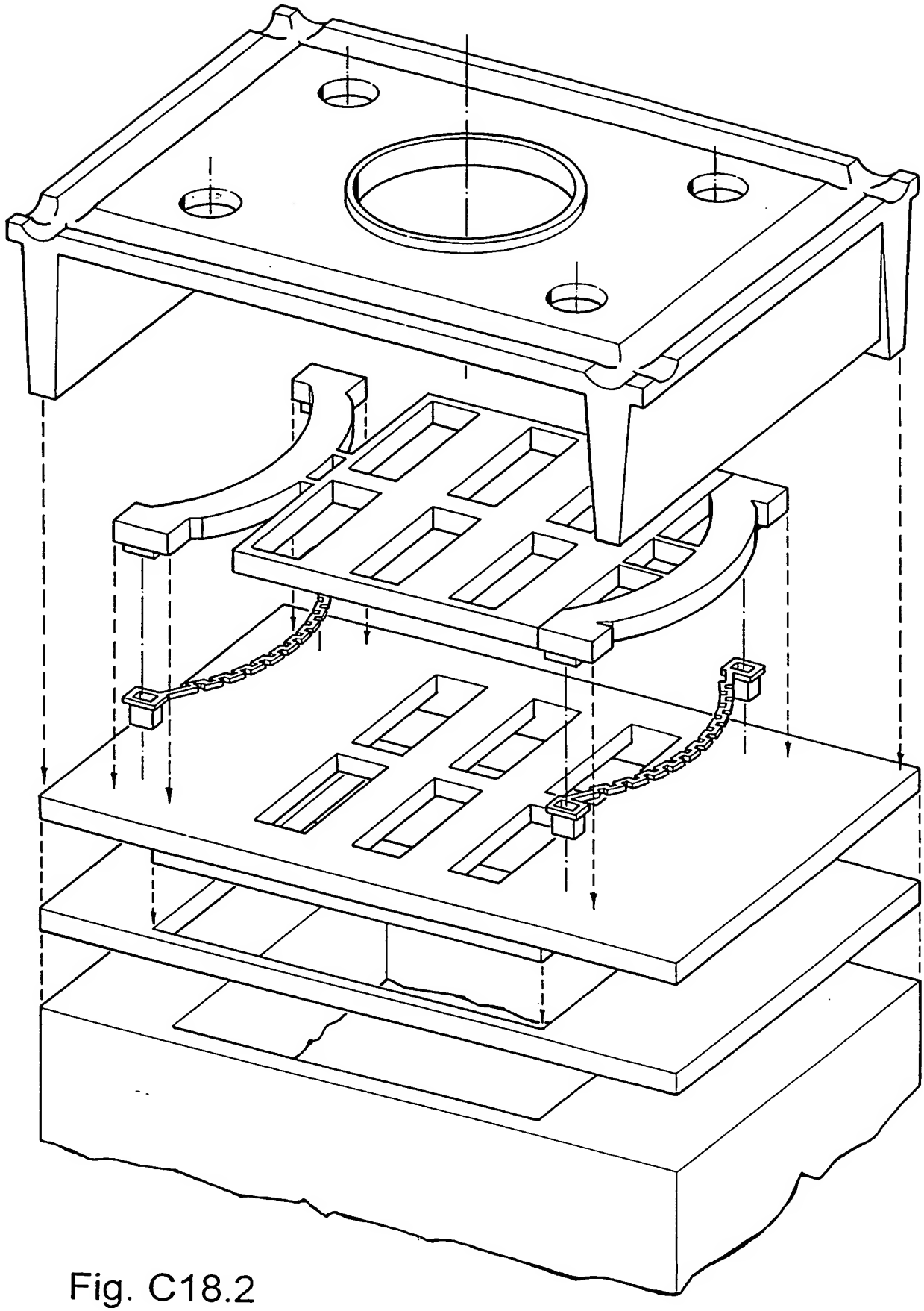


Fig. C18.2

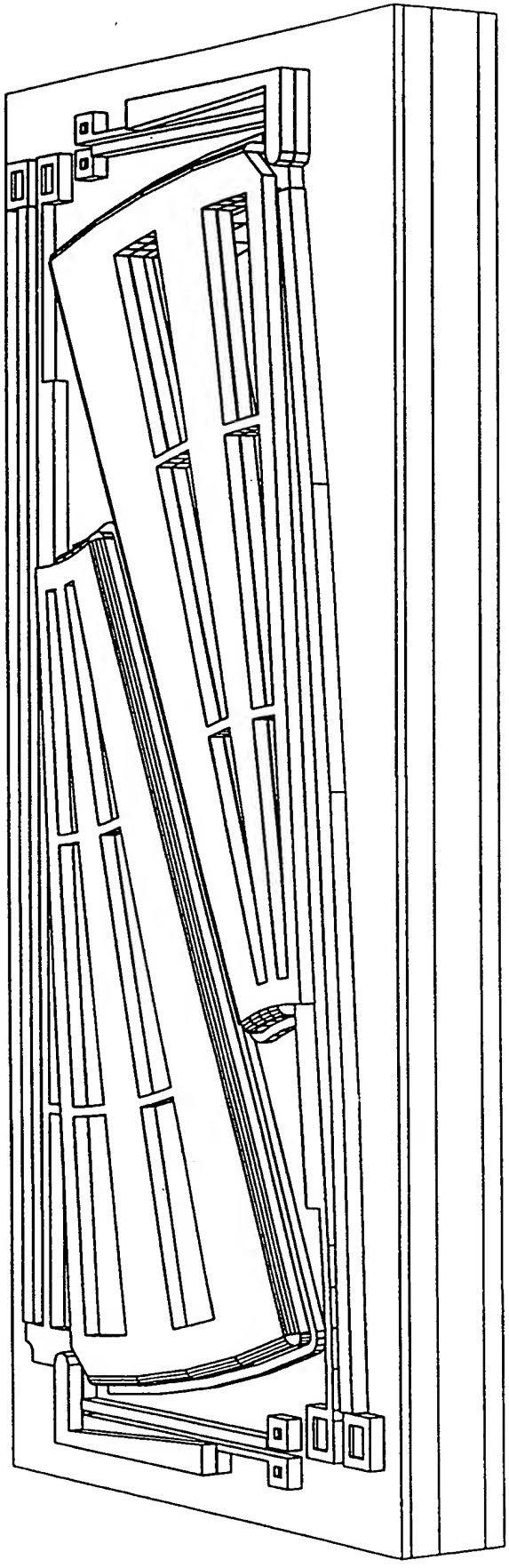


Fig. C19.1

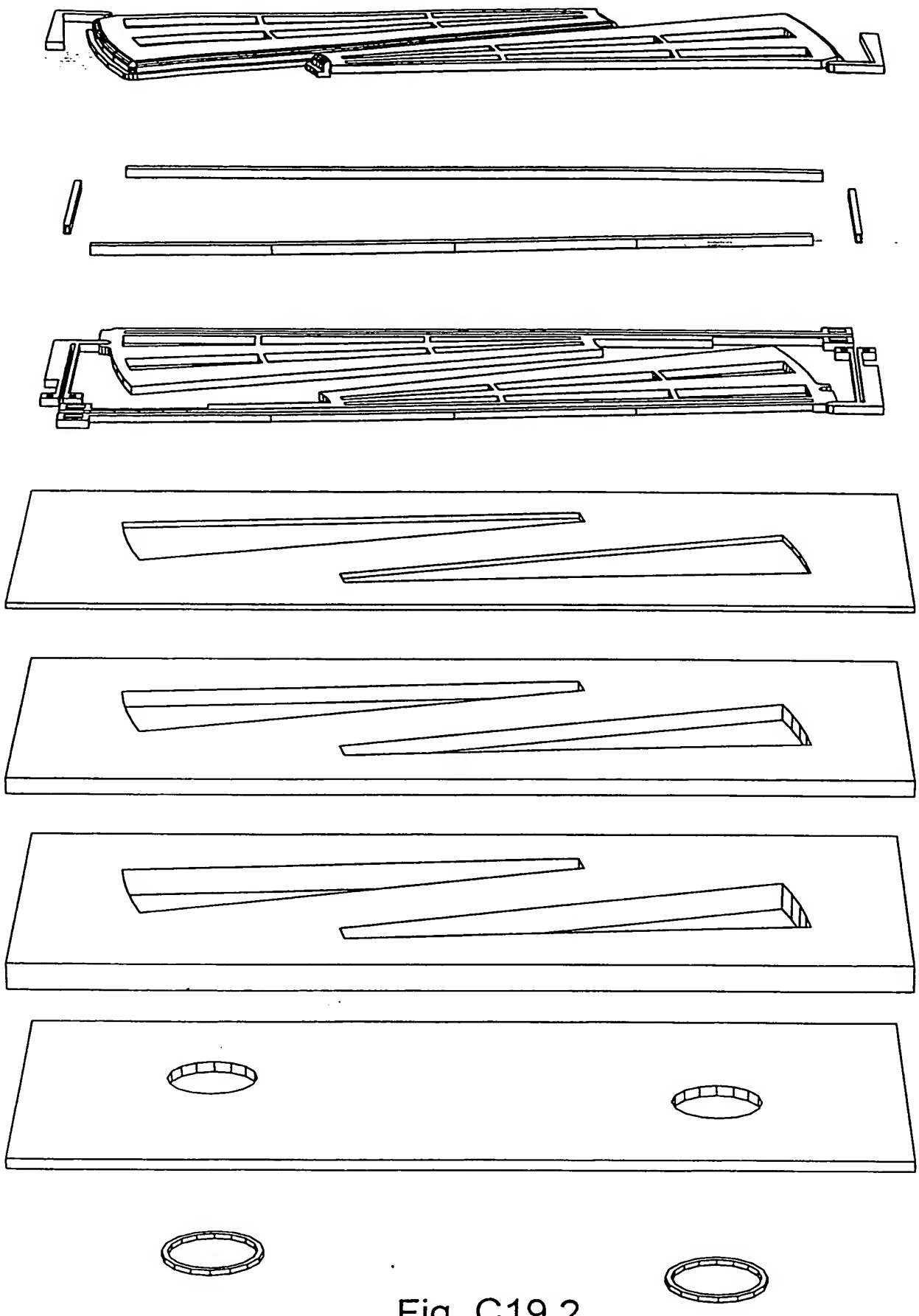


Fig. C19.2

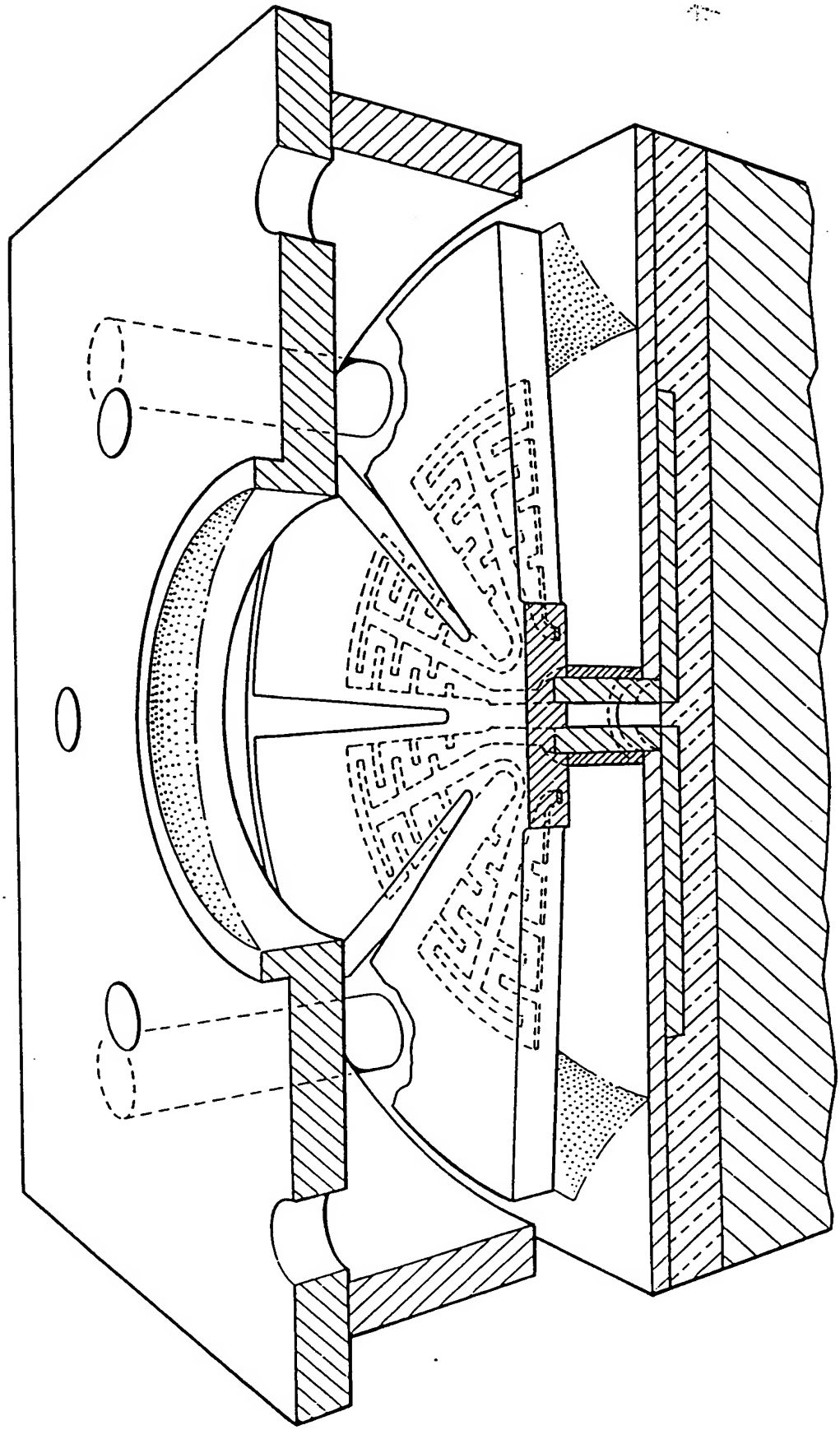


Fig. C20.1

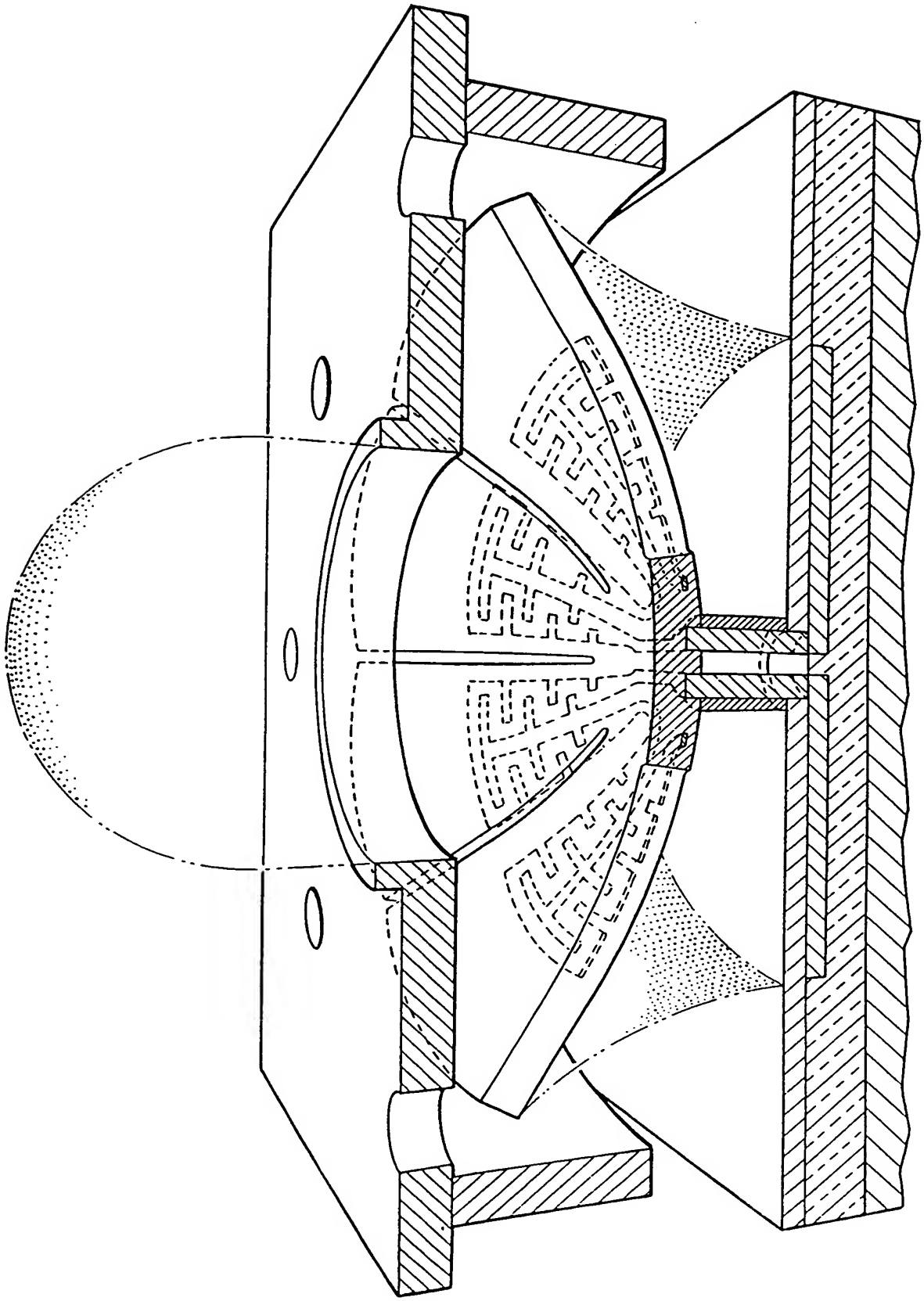


Fig. C20.2

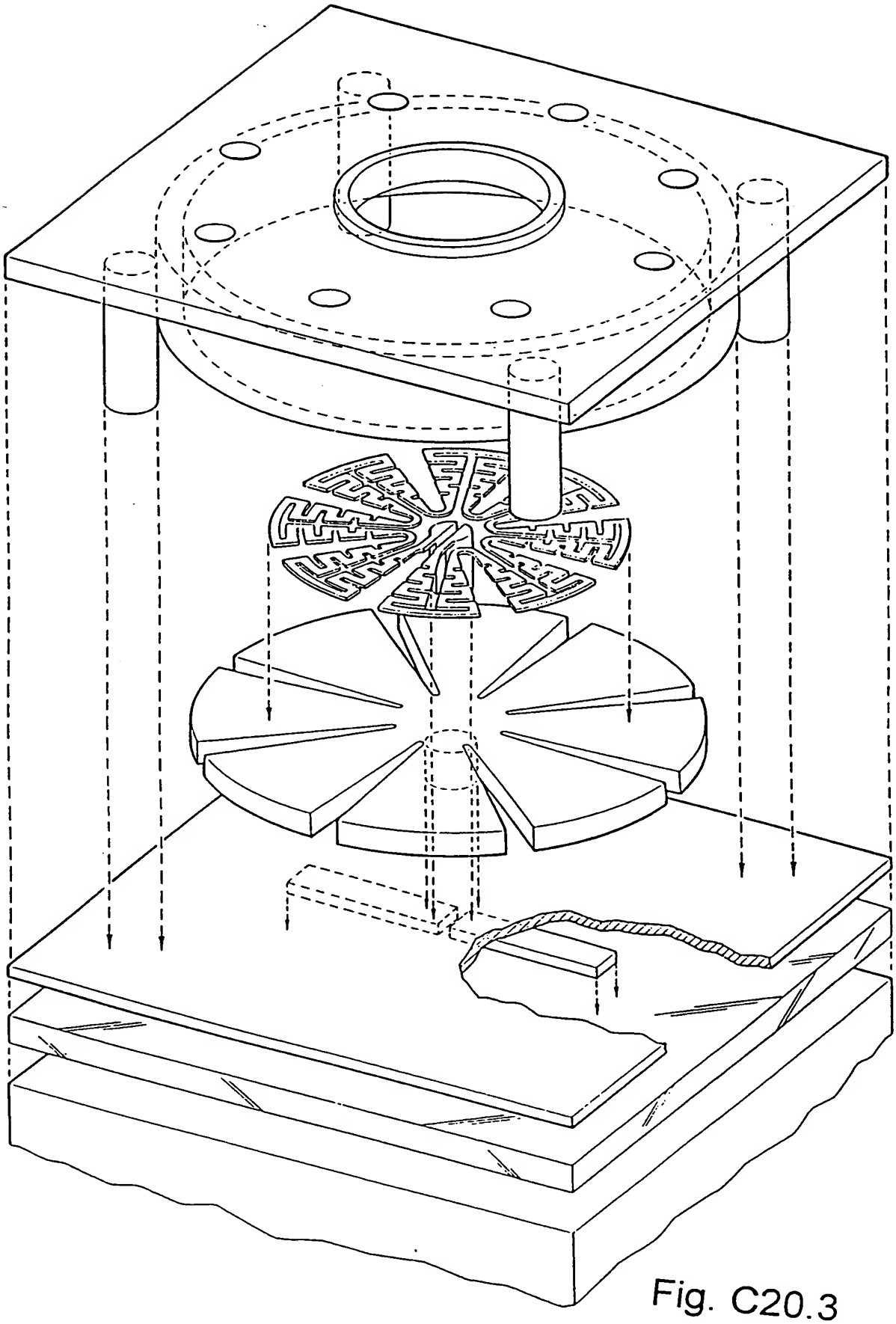


Fig. C20.3

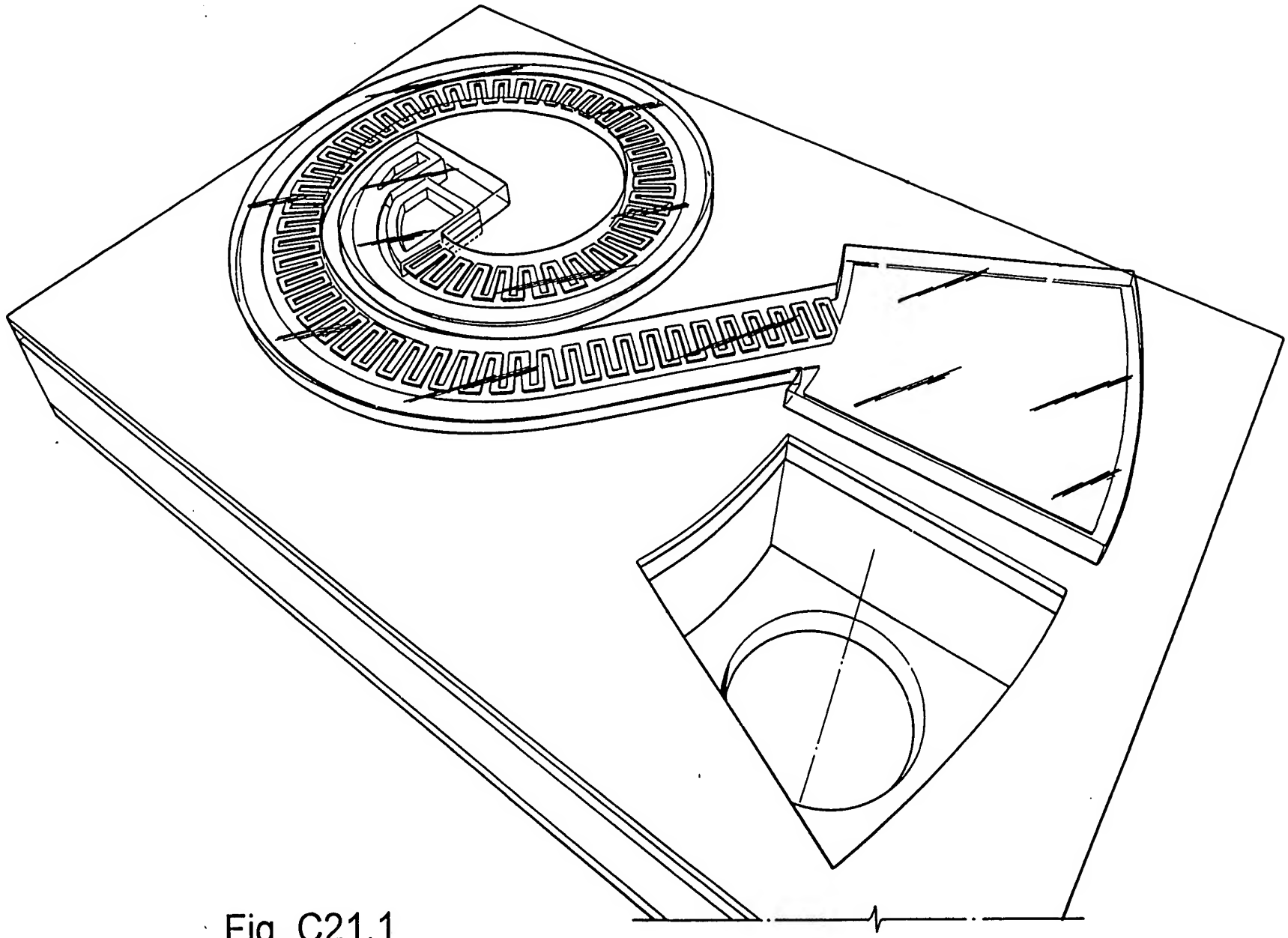


Fig. C21.1

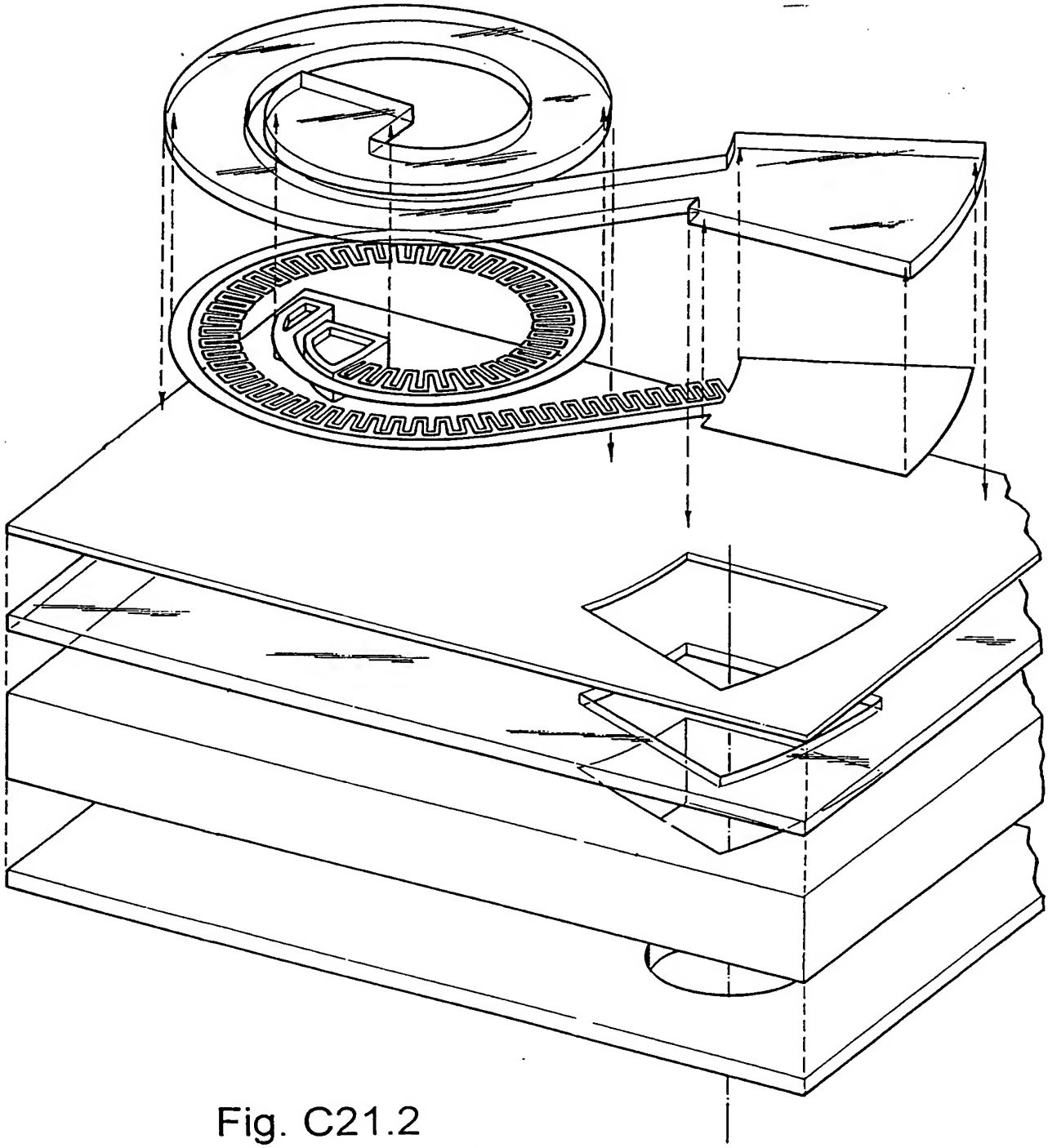


Fig. C21.2

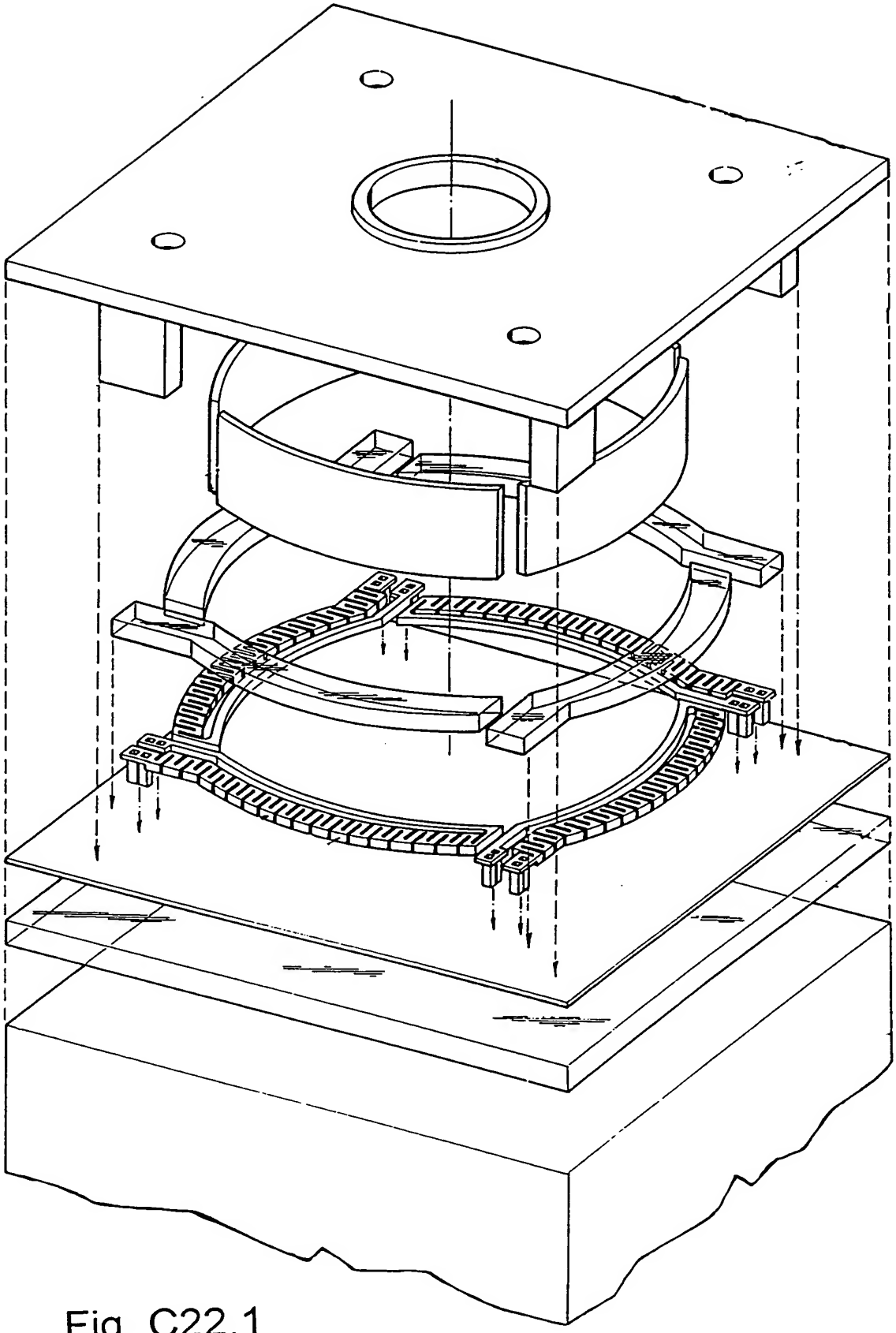


Fig. C22.1

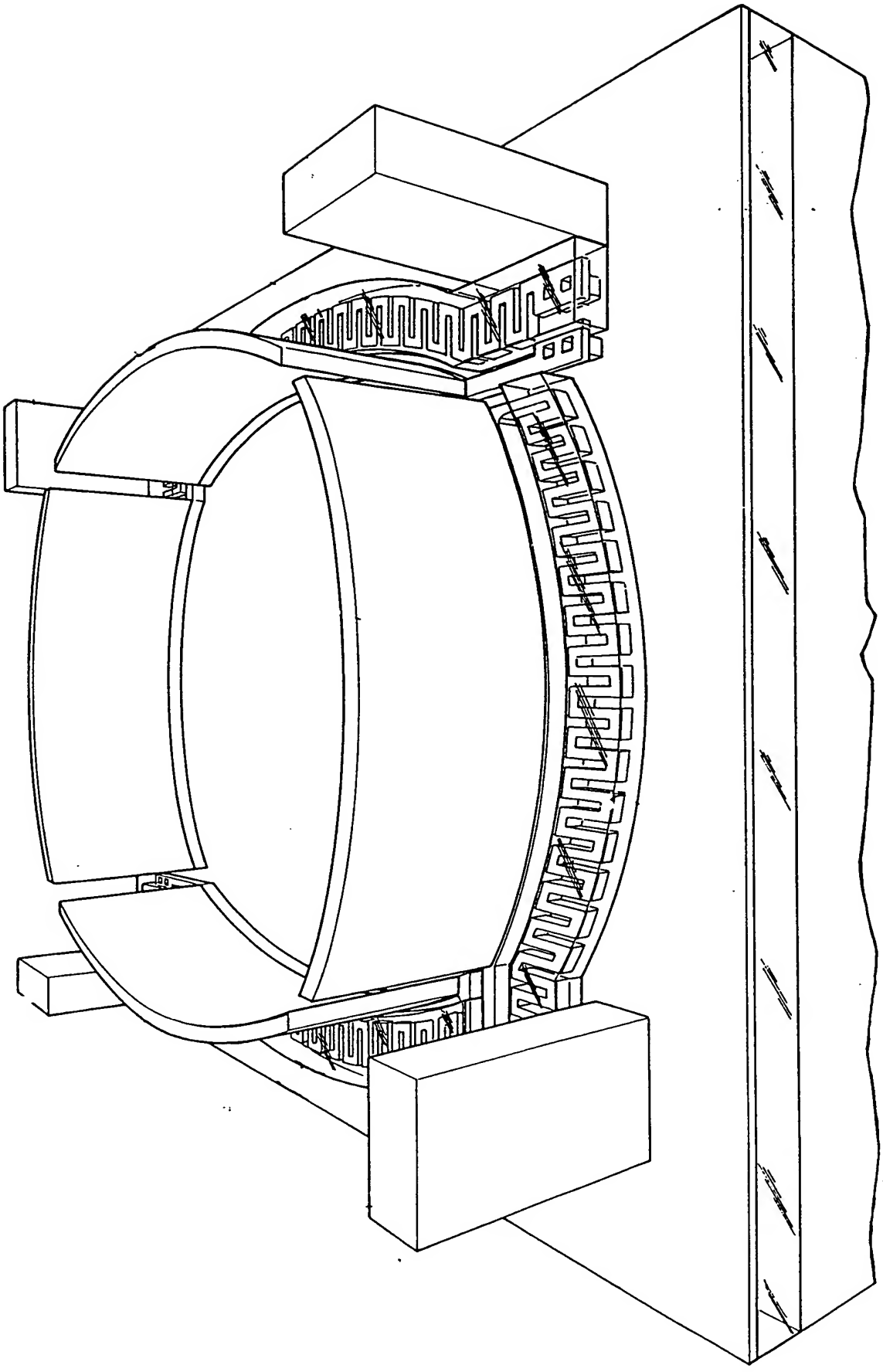


Fig. C22.2

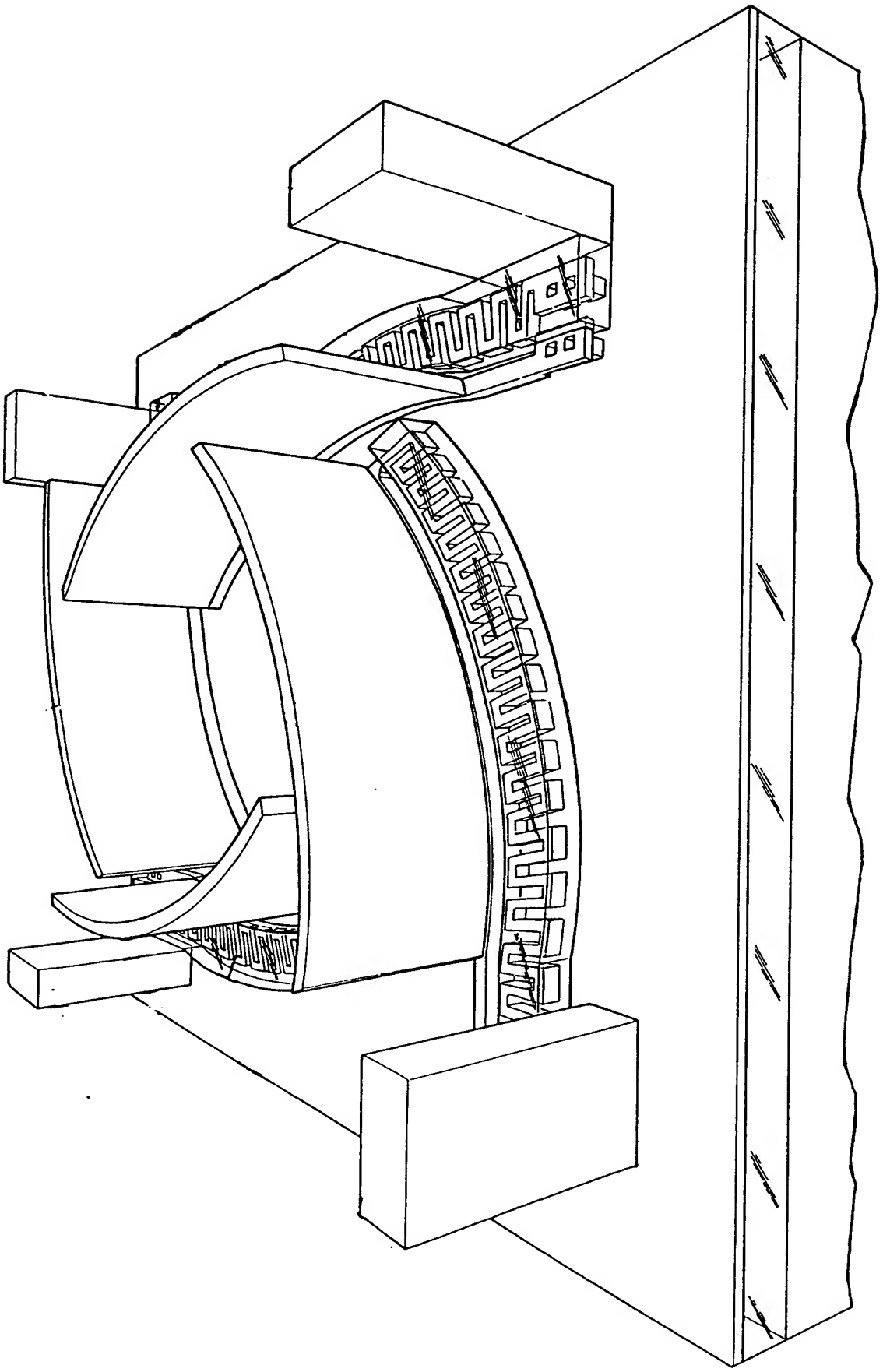


Fig. C22.3

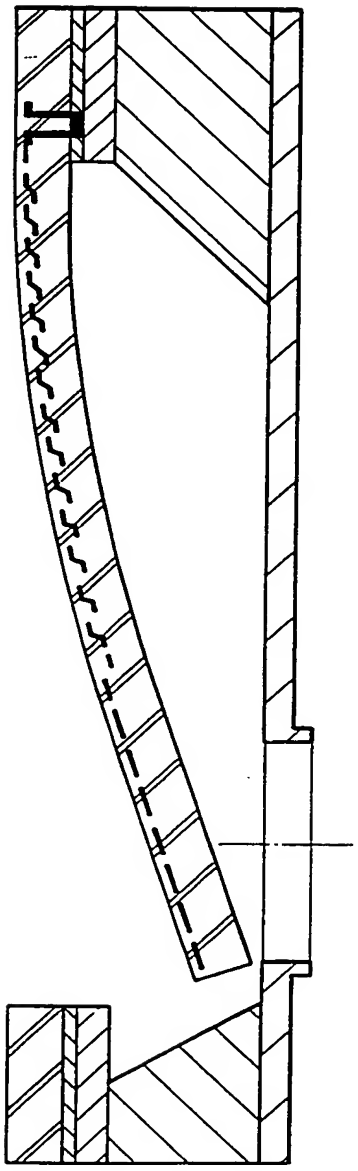


Fig. C23.1

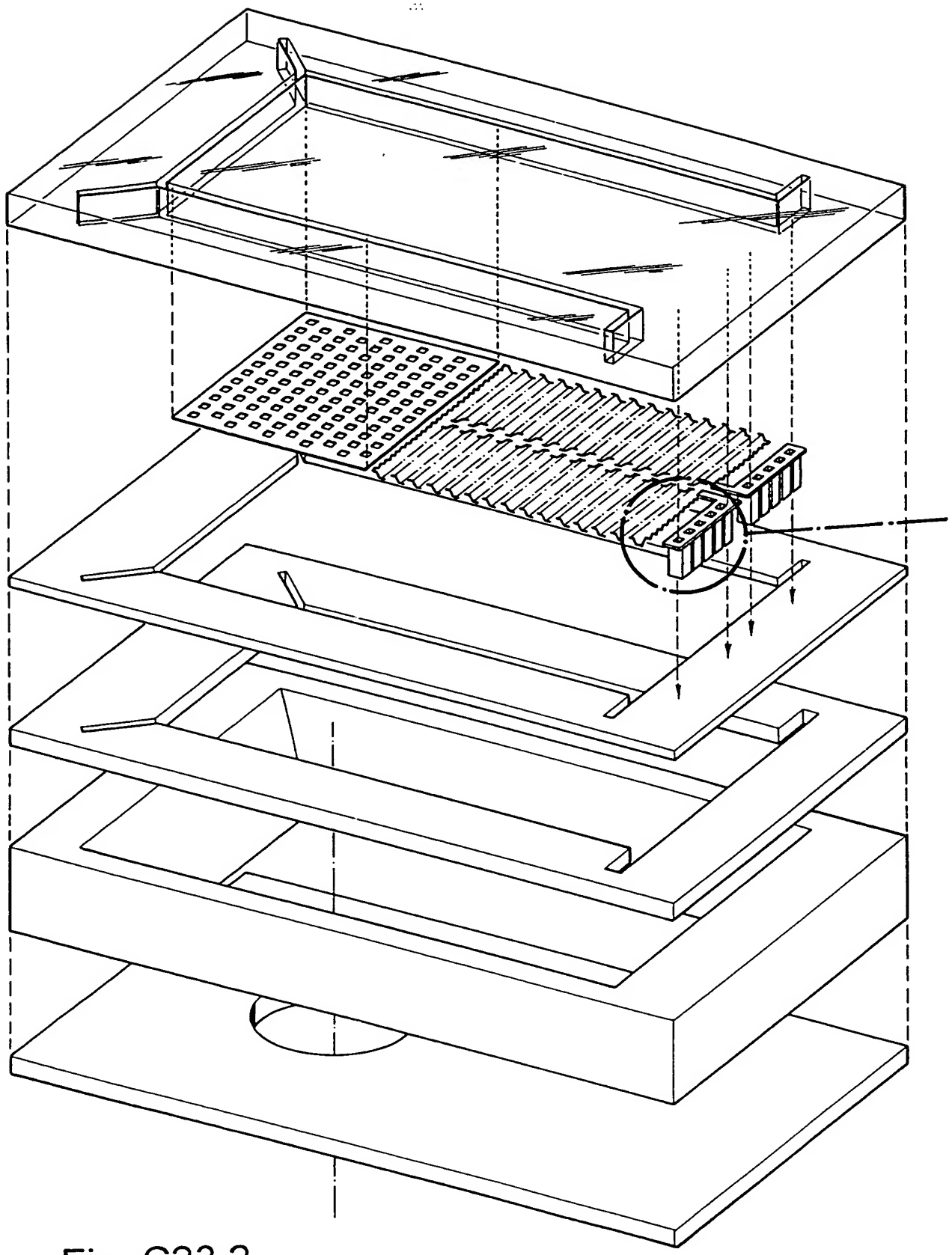


Fig. C23.2

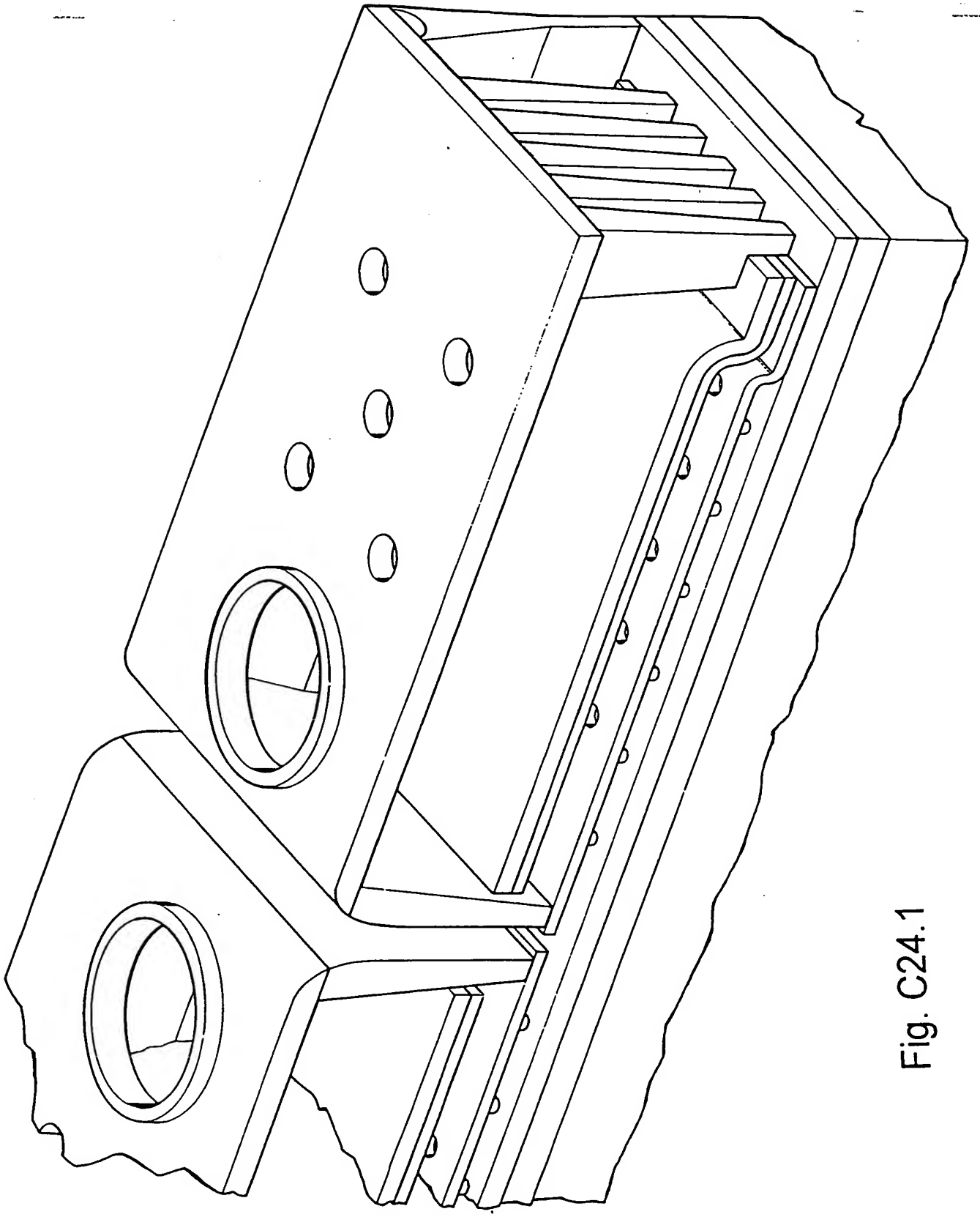


Fig. C24.1

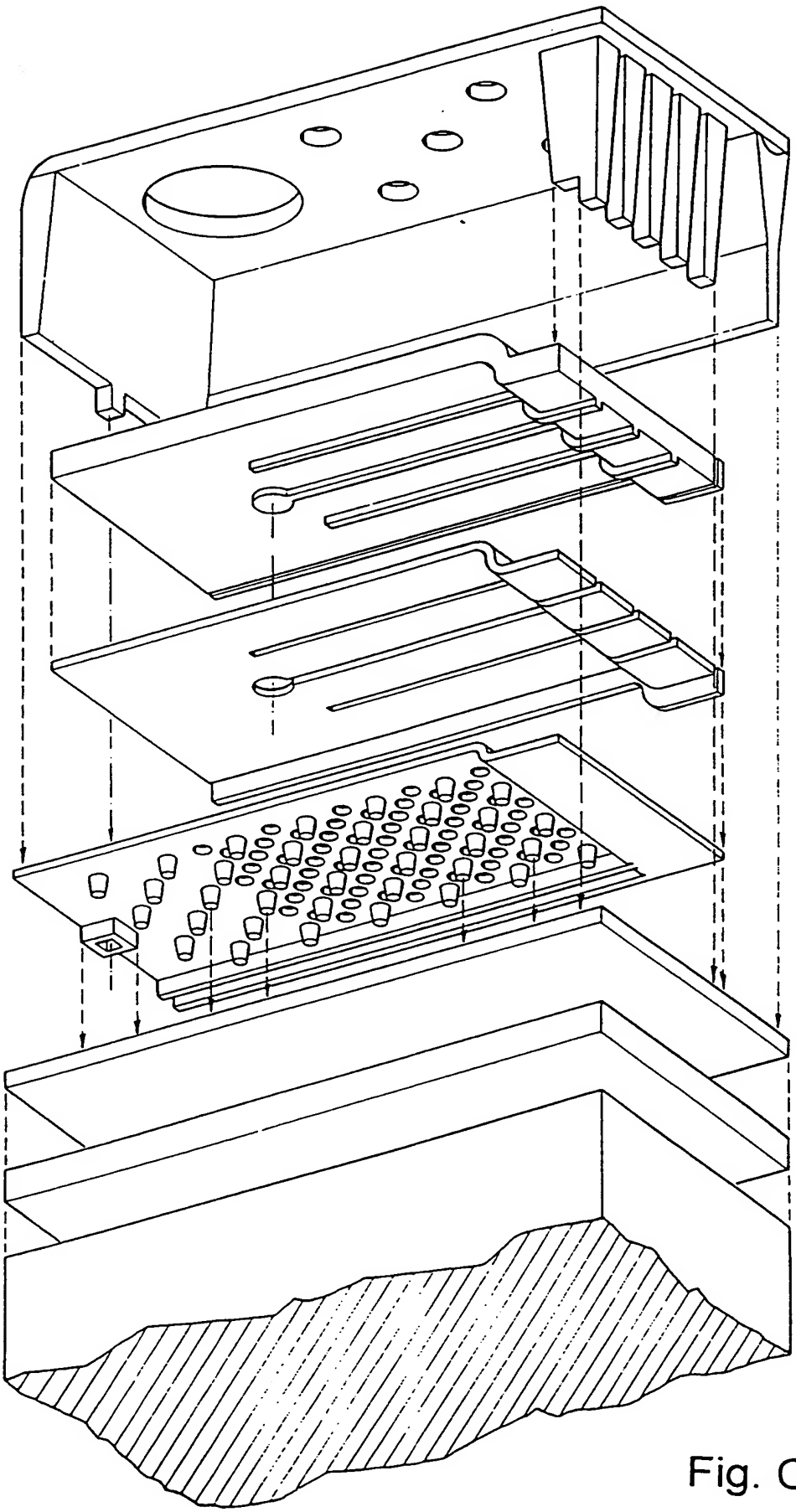


Fig. C24.2

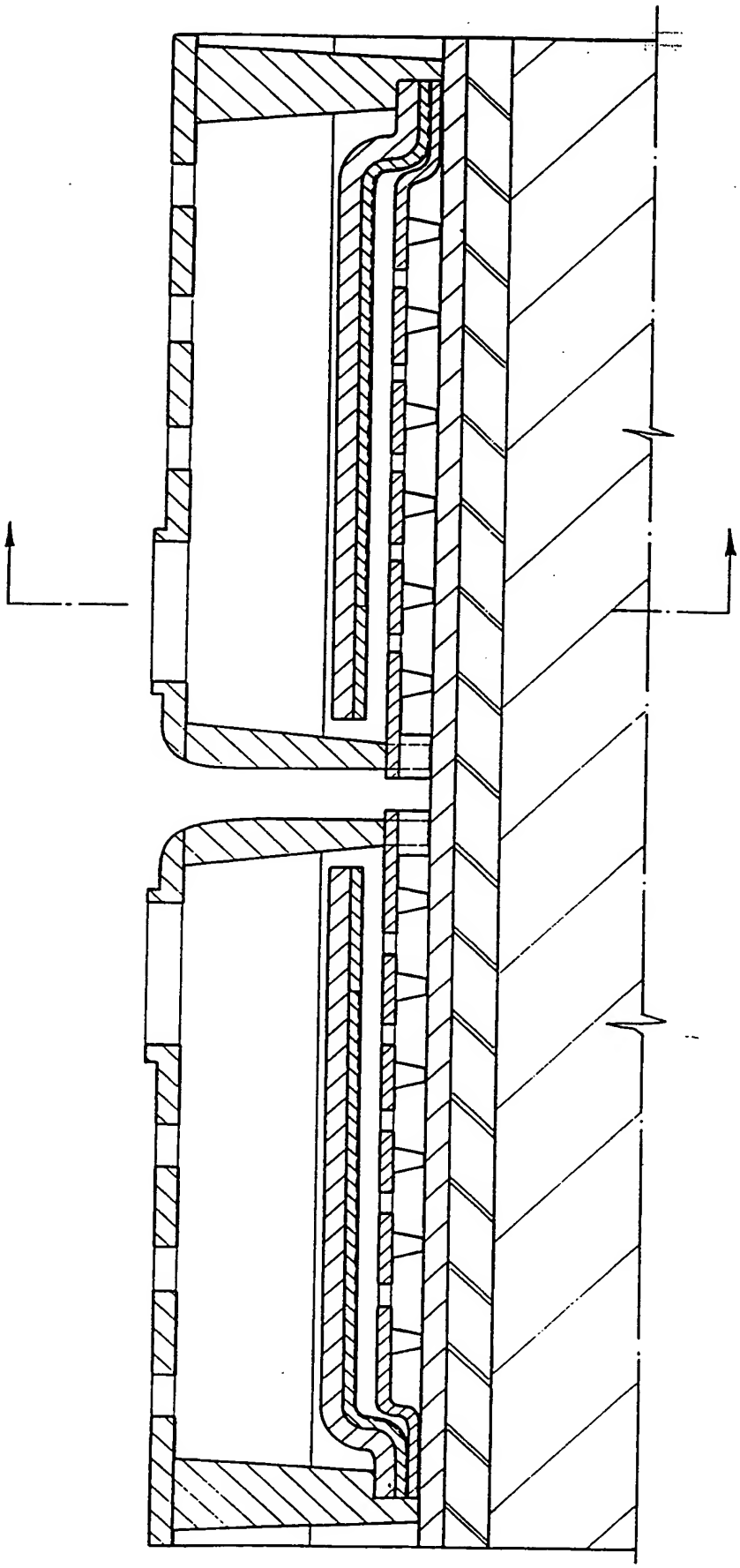


Fig. C24.3

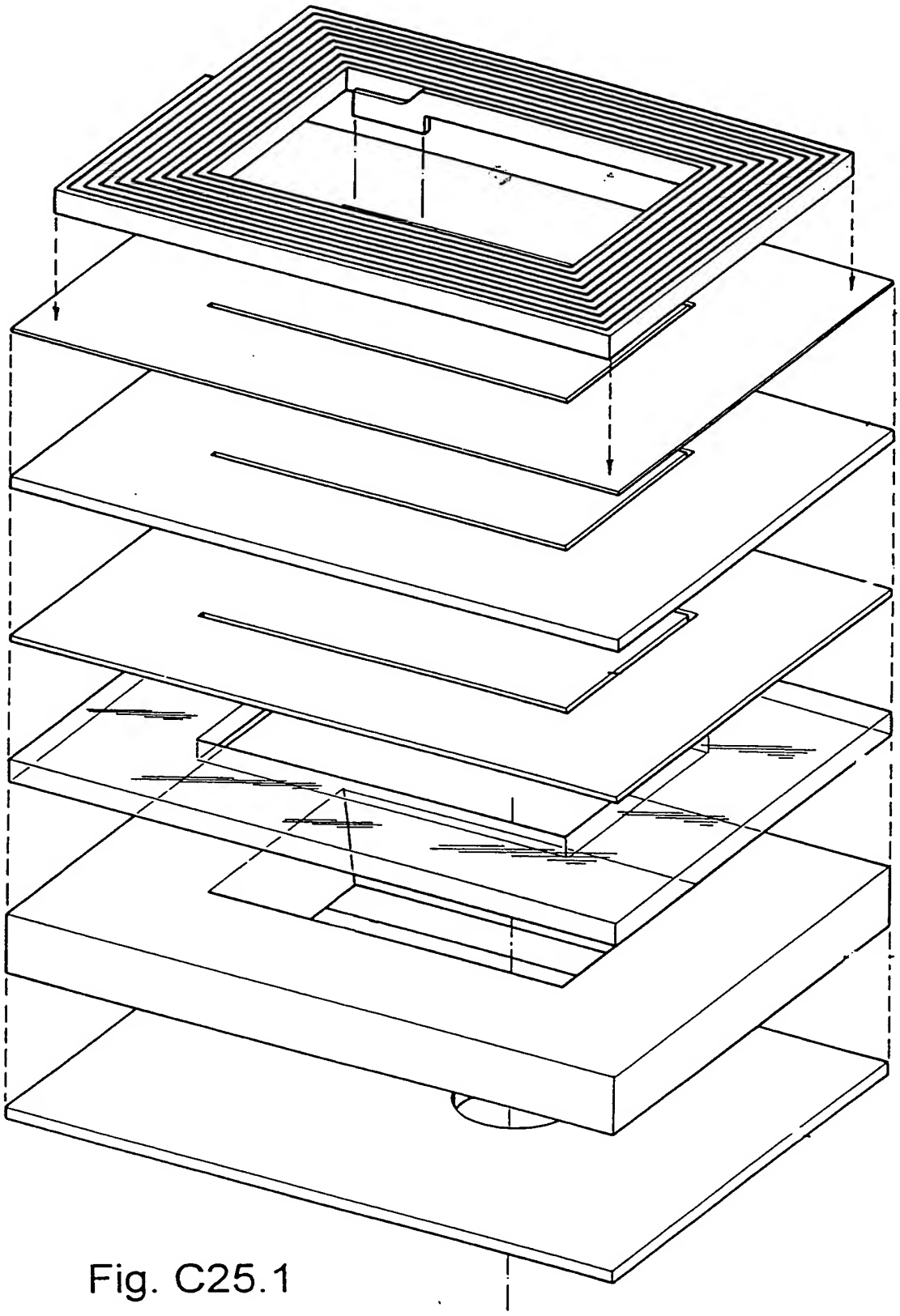


Fig. C25.1

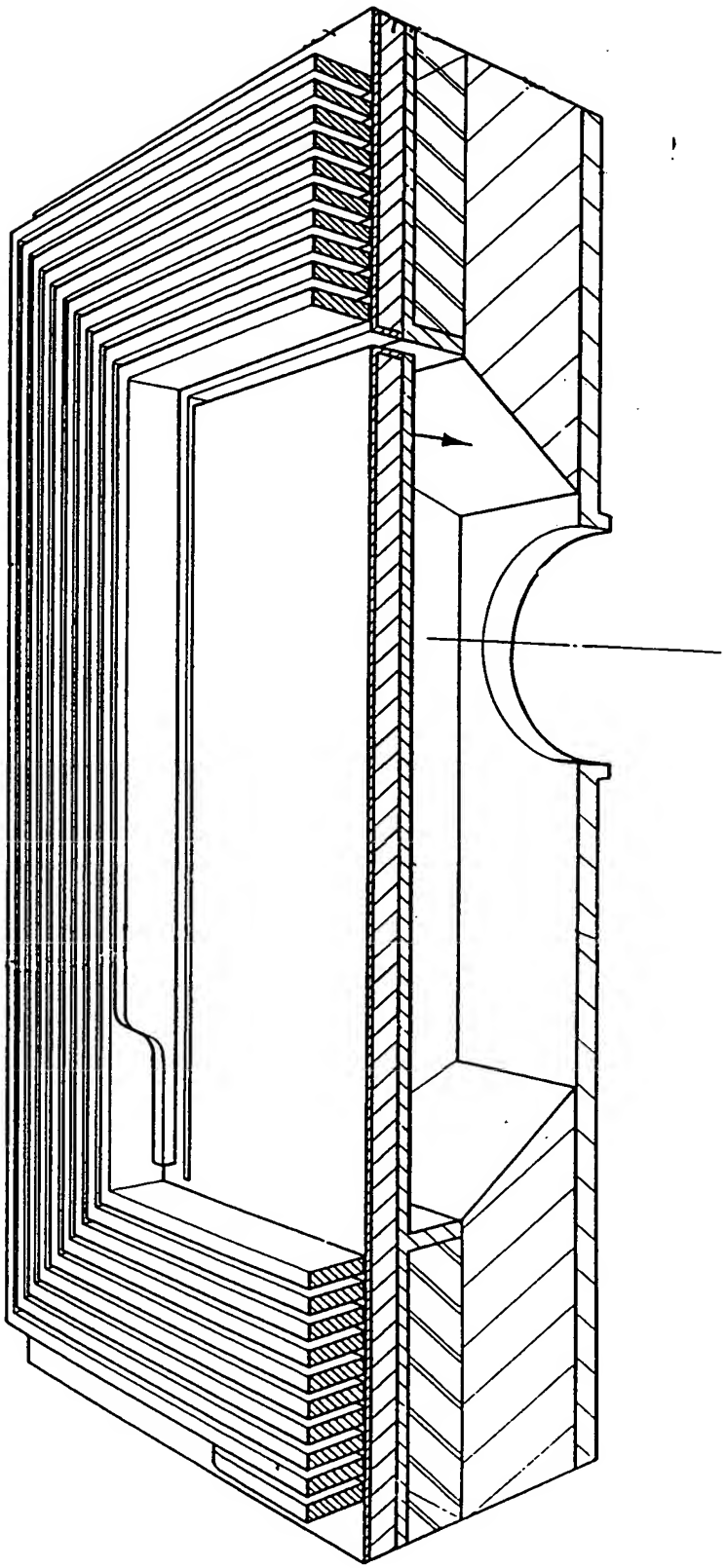


Fig. C25.2

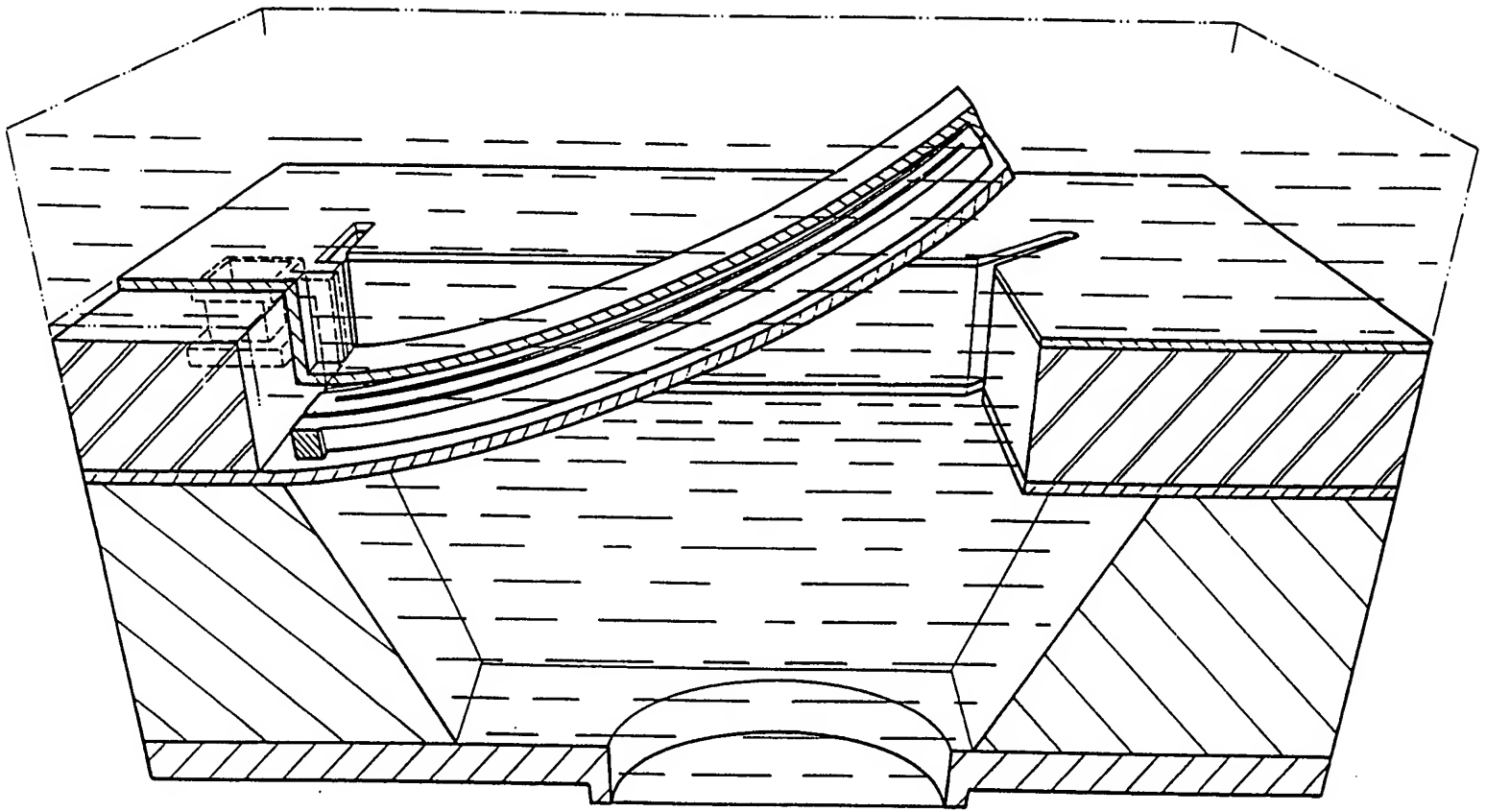


Fig. C26.1

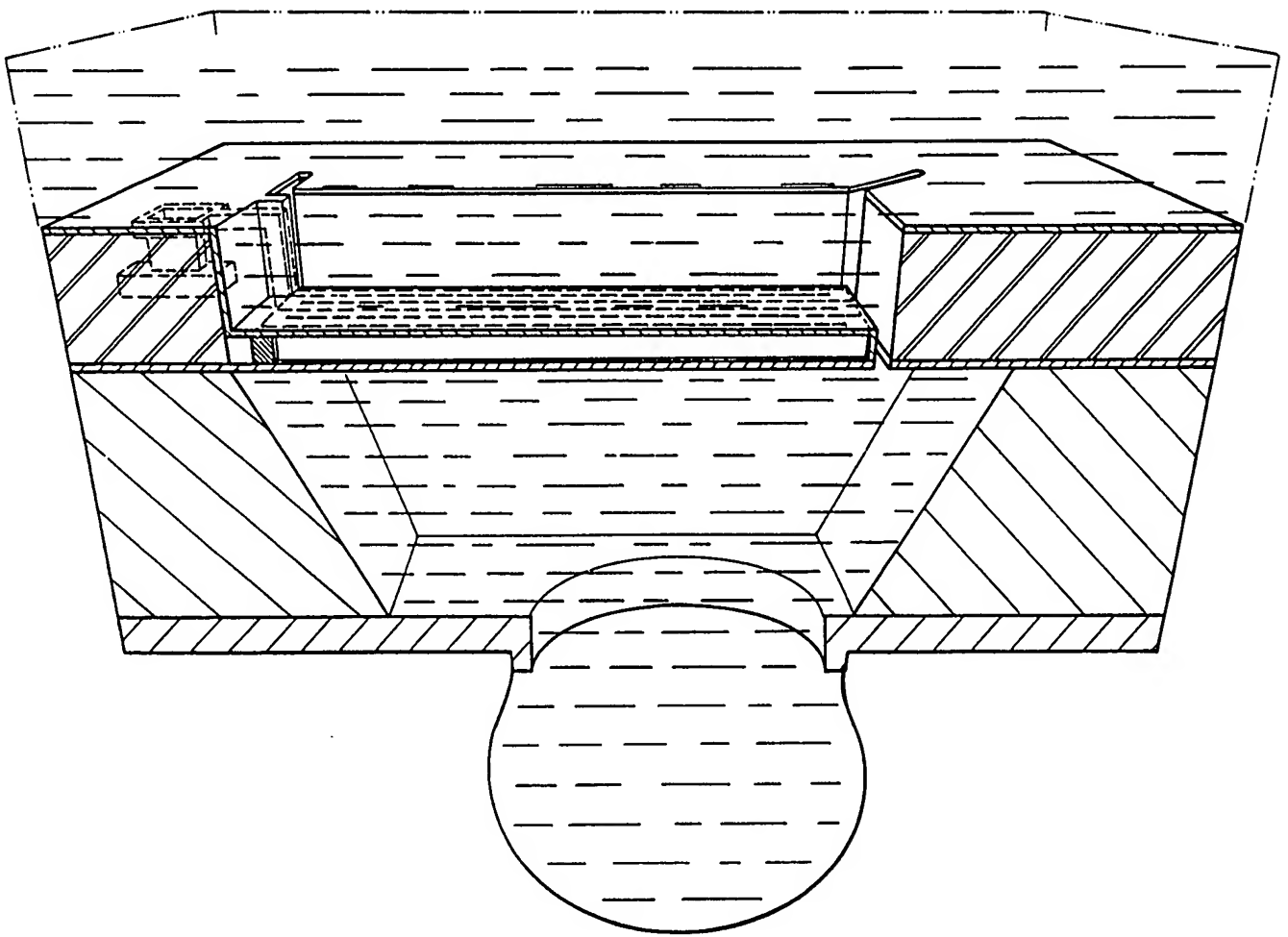


Fig. C26.2

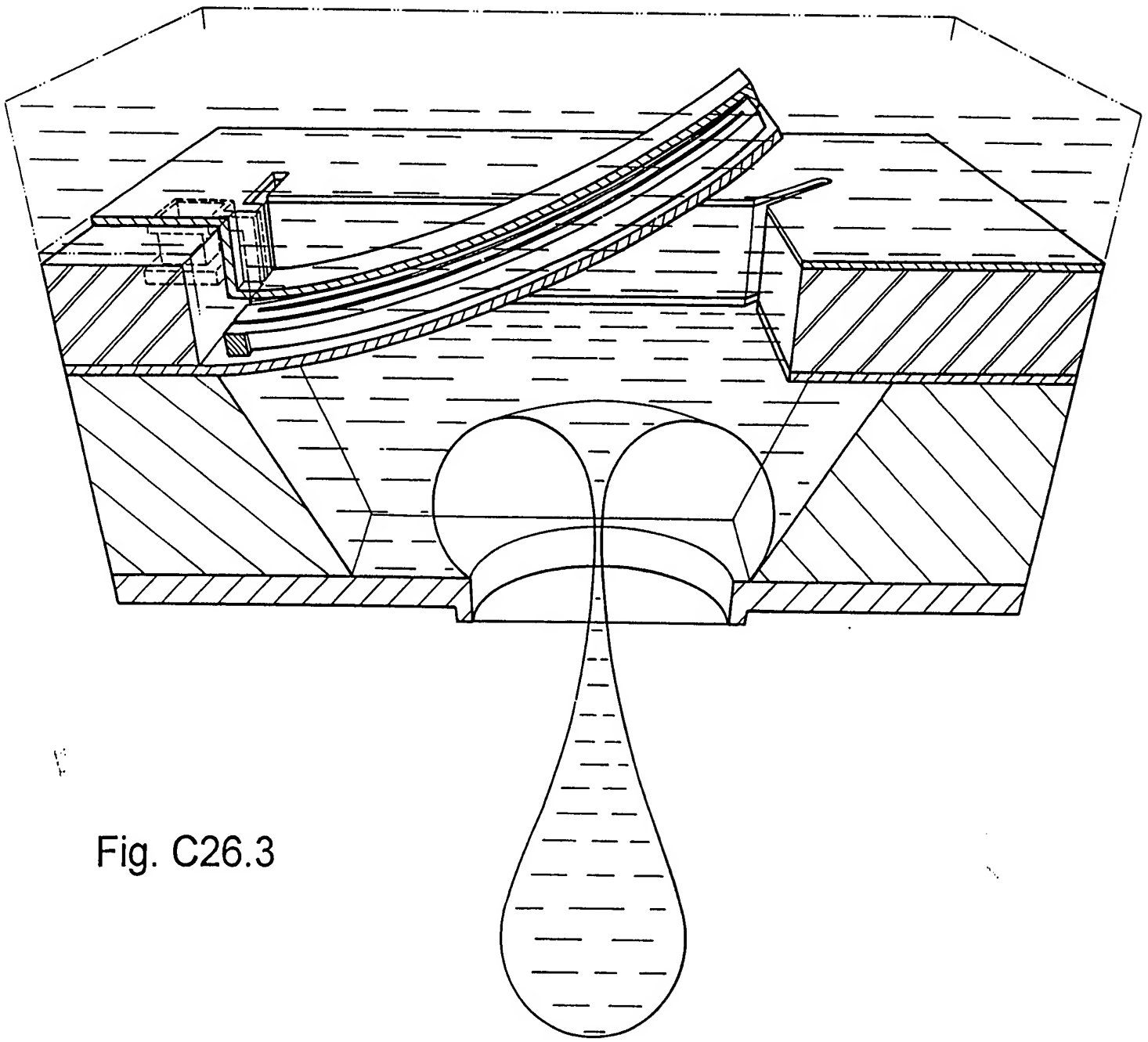


Fig. C26.3

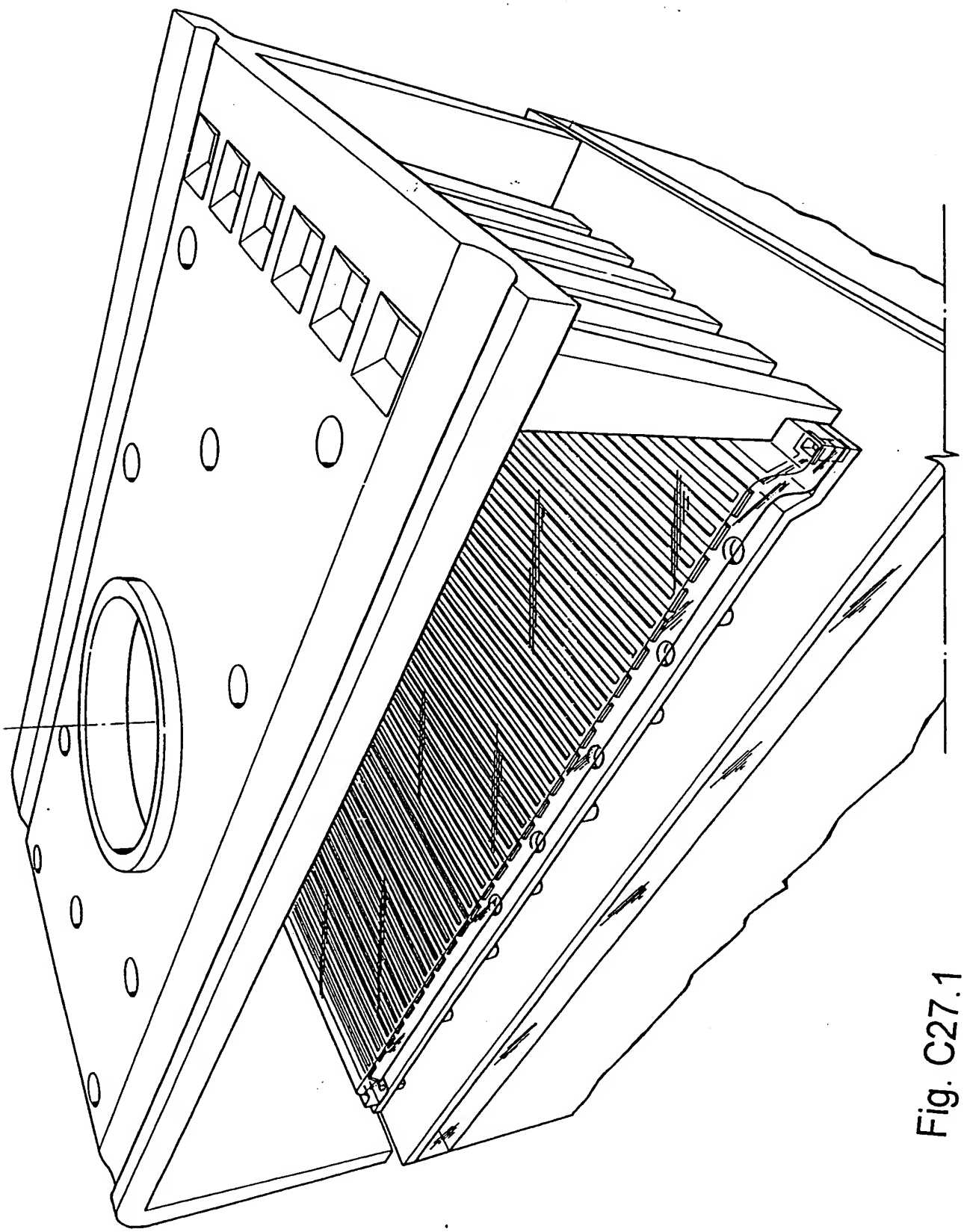


Fig. C27.1

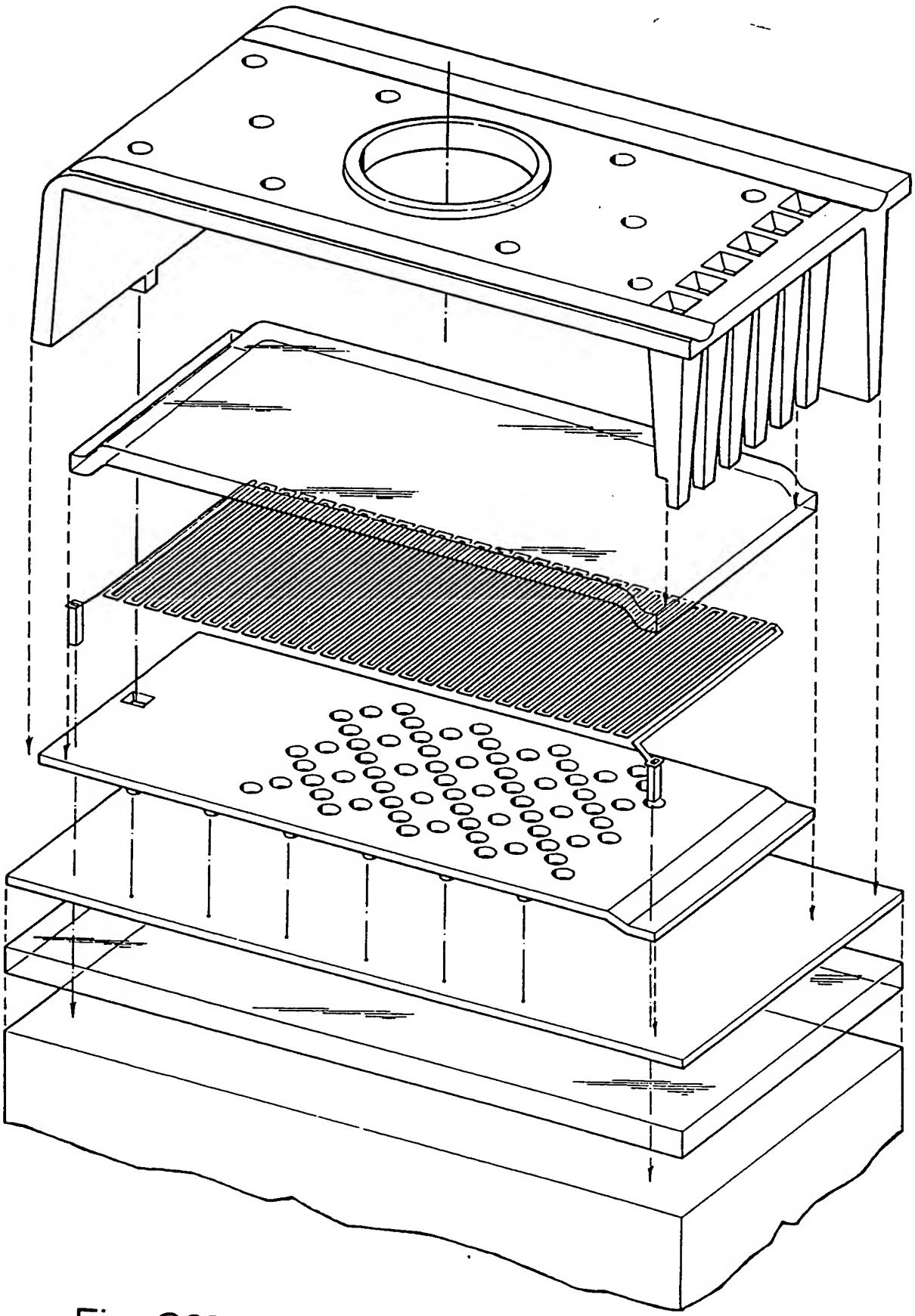


Fig. C27.2

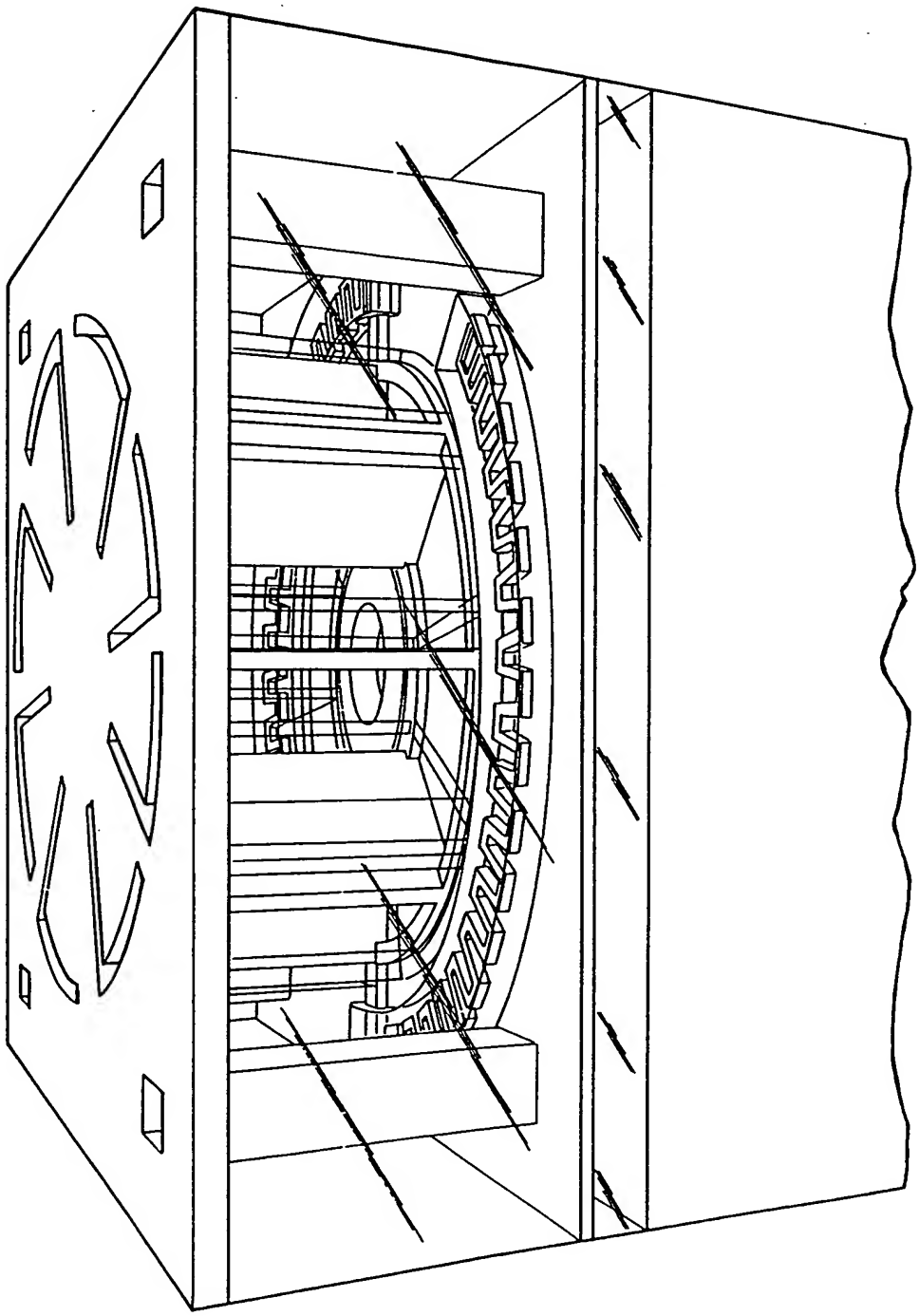


Fig. C28.1

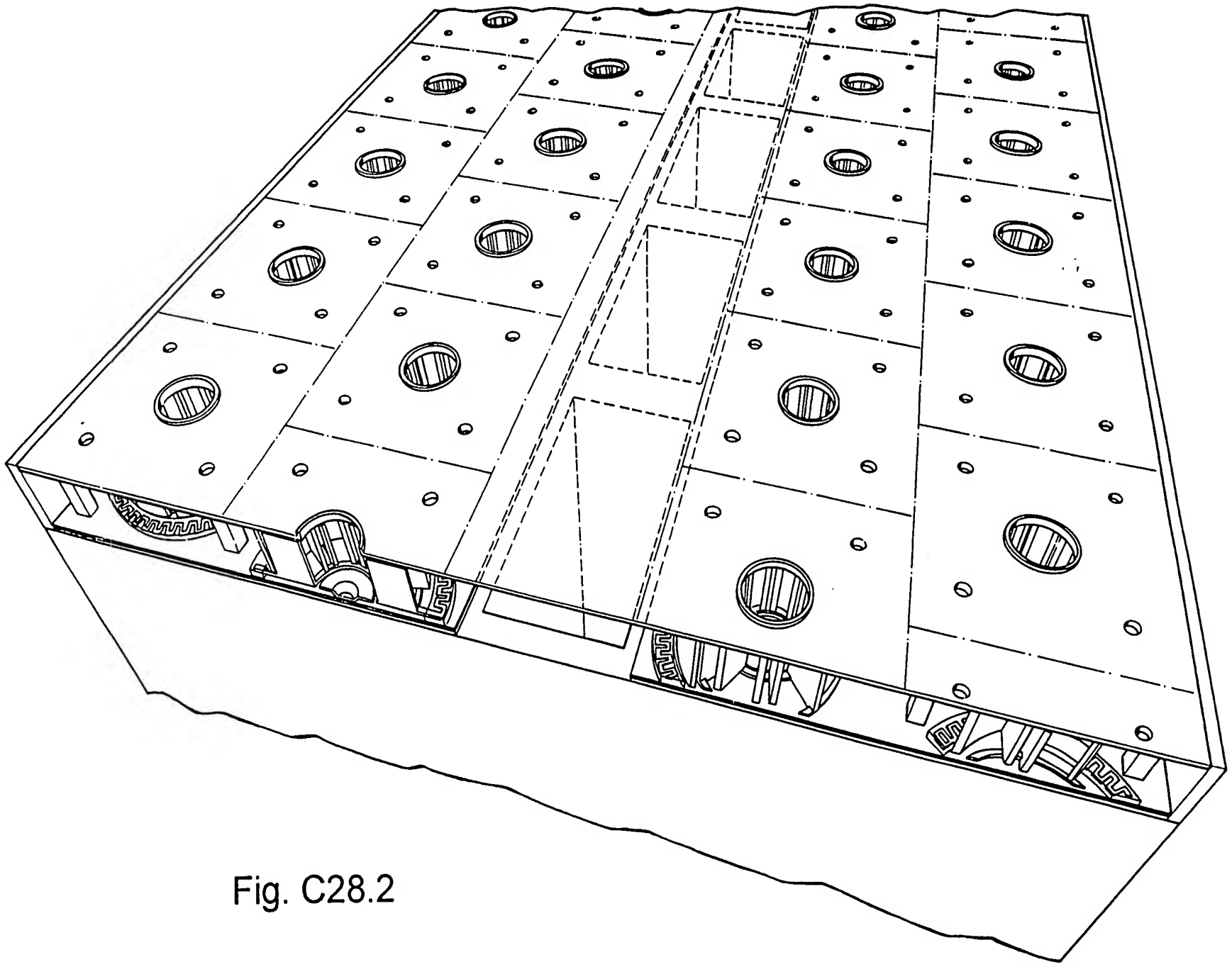


Fig. C28.2

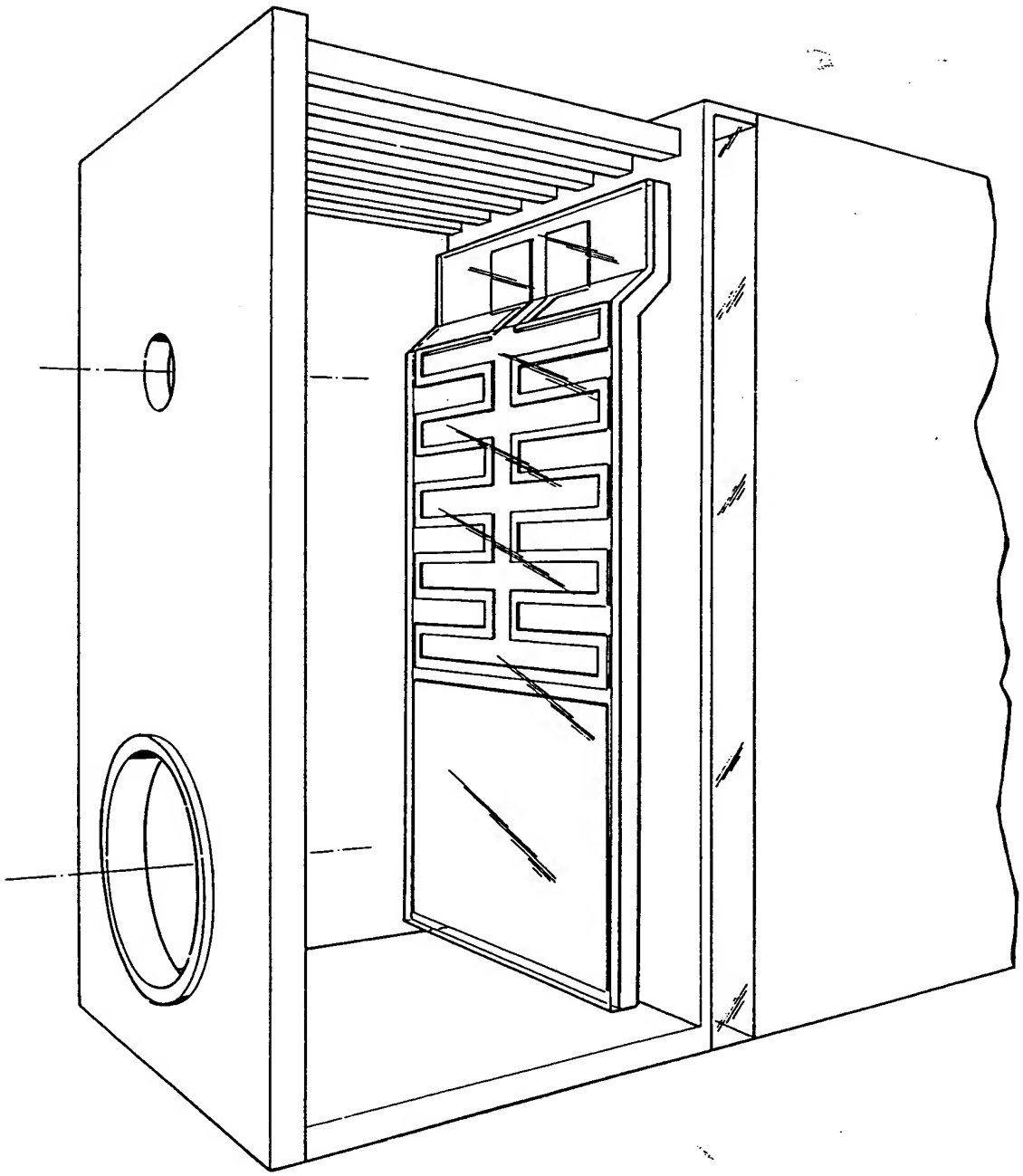


Fig. C29.1

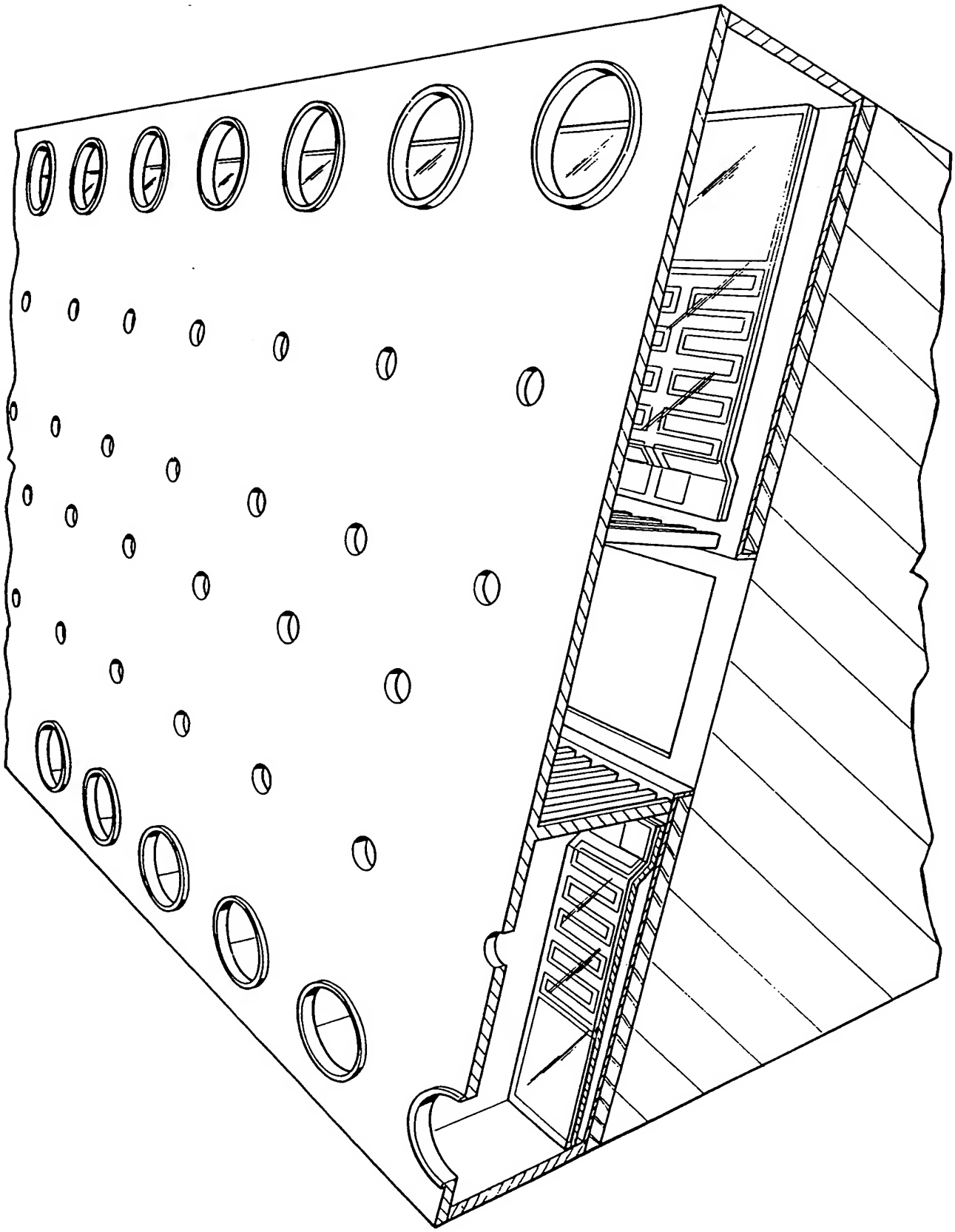


Fig. C29.2

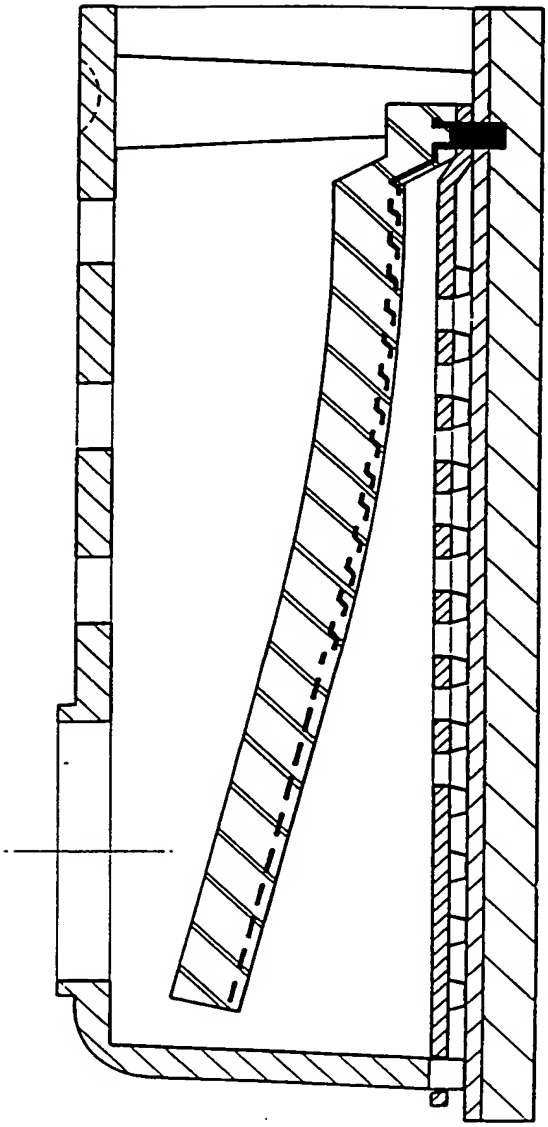


Fig. C30.1

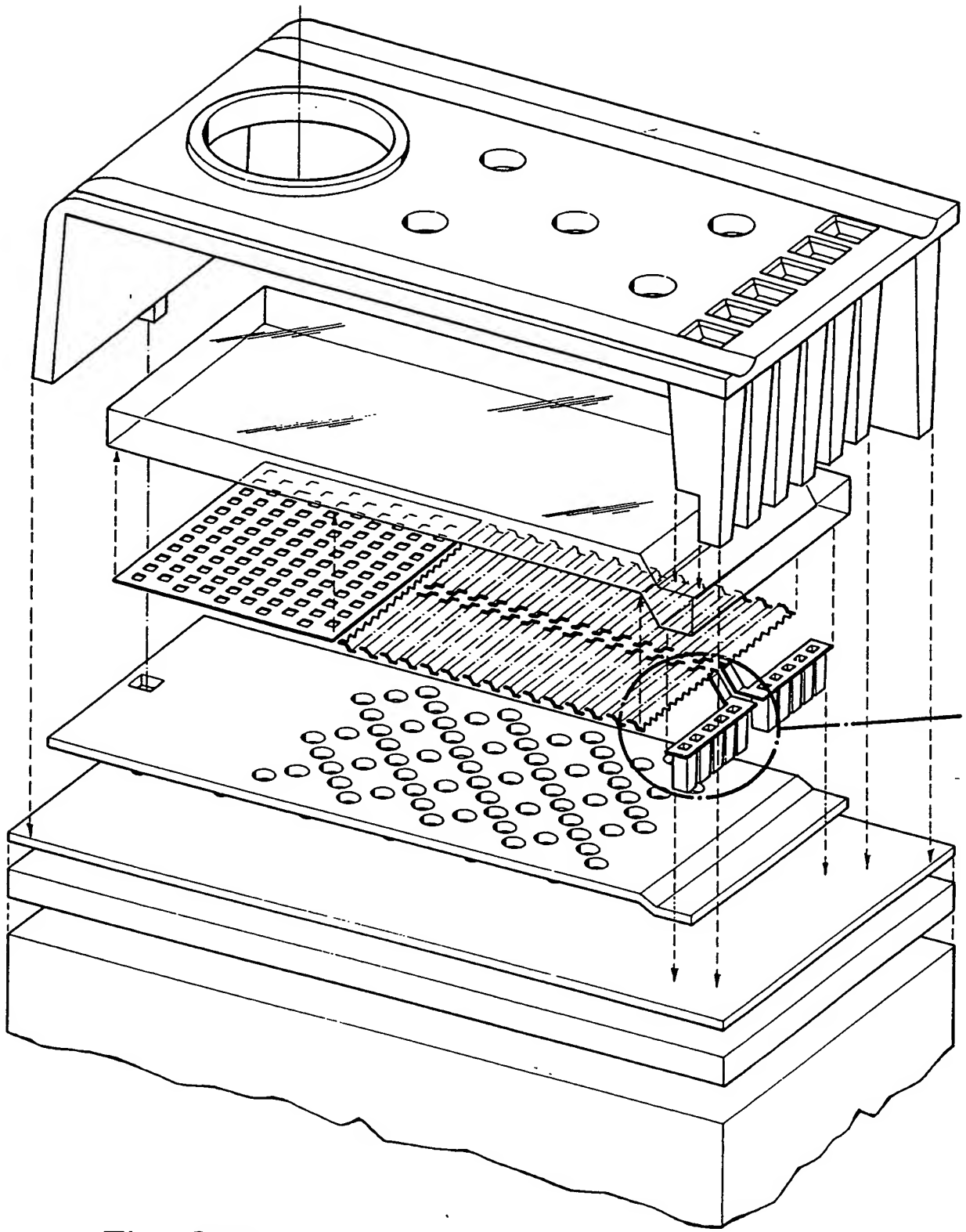


Fig. C30.2

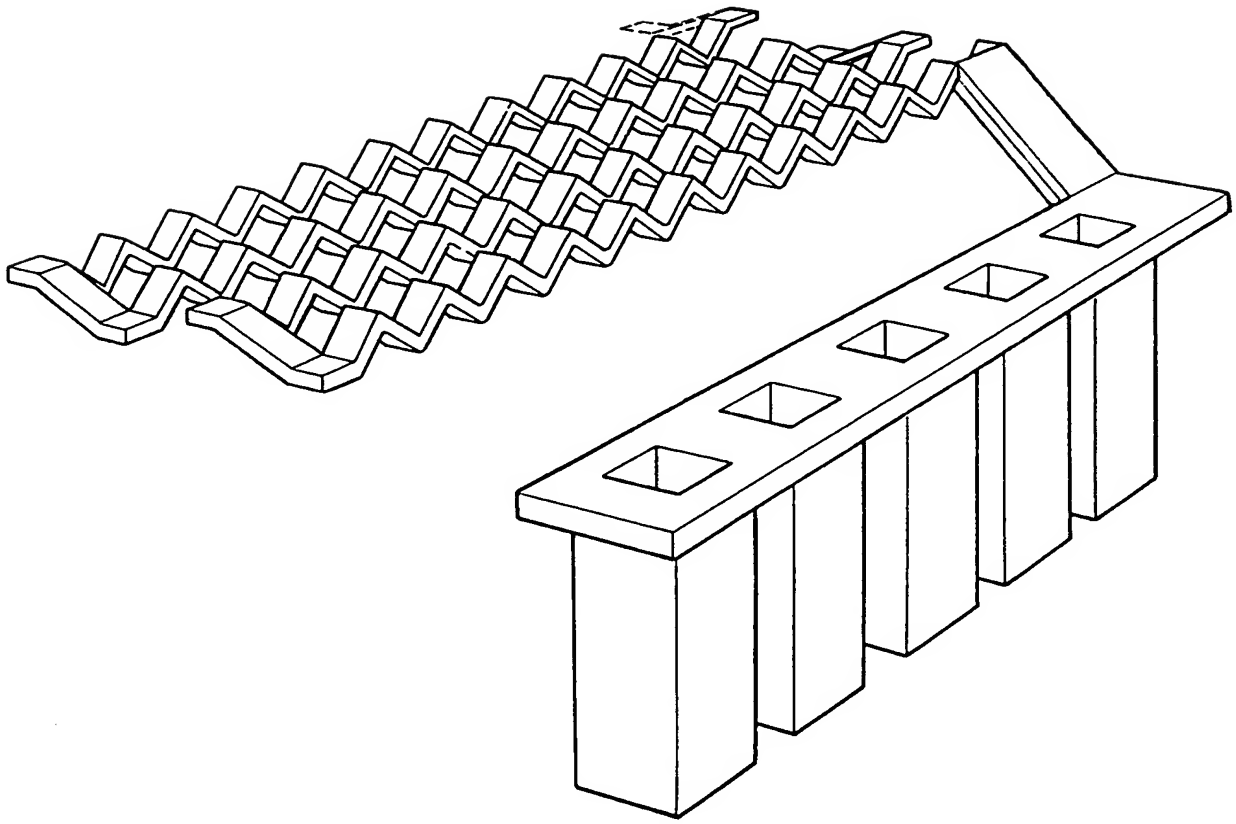


Fig. C30.3

THIS PAGE BLANK (USPTO)